Scheduling for PPCG

Sven Verdoolaege *Gerda Janssens*

Report CW 706, June 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Scheduling for PPCG

Sven Verdoolaege[*] *Gerda Janssens*

Report CW 706, June 2017

Department of Computer Science, KU Leuven

Abstract

The main purpose of this report is to provide a detailed description of the scheduler used by PPCG. In order to provide some context, the report starts with an overview of PPCG and a summary of how it performs dependence analysis, including a description of the interface to the isl dependence analysis engine and the way this interface is used by PPCG. The report then explains the interface to the isl scheduler and how it is used by PPCG, including a description of the way the results of the dependence analysis are used as input to the scheduler as well as of the further processing that is performed on the resulting schedule tree.

The core of the report is formed by a detailed description of the forced outer coincidence strategy as a variation of the Pluto scheduler with Feautrier as fallback. Both core schedulers along with their implementation details in isl are described. Finally, some known issues with these schedulers are illustrated and solutions are presented for most of these issues. These include incremental scheduling, measures for avoiding and/or preventing loop coalescing schedules and considering self-dependences first in the Feautrier scheduler.

[*]Part of the work described in this report was performed while Sven Verdoolaege was working for École Normale Supérieure and for INRIA Paris.

Contents

1 Introduction

PPCG (Verdoolaege, Juega, et al. 2013) is a source-to-source polyhedral parallelizing compiler that is mainly targeted to the generation of CUDA (Nvidia 2011) or OpenCL code (Stone et al. 2010). As a polyhedral compiler, PPCG extracts a polyhedral model (Feautrier and Lengauer 2011) from the input code, analyzes and transforms this model and finally generates code again from the model. One of the transformation steps is the computation of an affine schedule (Feautrier 1992b), which is used to detect and expose the available parallelism.

This report describes some details of the scheduler implemented in isl (Verdoolaege 2010) and used by PPCG. This scheduler was first described, if briefly, by Verdoolaege, Juega, et al. (2013, Section 6). An extension to live-range reordering was described by Verdoolaege and Cohen (2016). In order to make the report more self-contained, it also provides an overview of the inner workings of PPCG. Unless otherwise notes, this report describes the state of ppcg-0.07-13-ge61fb392.

Some of the test cases that illustrate various features of the scheduler are taken from PolyBench/C (Pouchet 2012; Pouchet and Yuki 2015). Whenever such a benchmark is used (except when explicitly stated otherwise), the command line option -DPOLYBENCH_USE_C99_PROTO is specified since without this macro being defined, the benchmarks are inconsistent, having fixed size arrays but parametrically bounded loops iterating over them.

2 Overview of PPCG

This section provides a brief overview of PPCG. More details are available from Verdoolaege, Juega, et al. (2013), although the latter description does not take into account more recent developments.

PPCG takes PENCIL code as input and produces CUDA or OpenCL as output. PENCIL is essentially C99 with some restrictions and with some additional

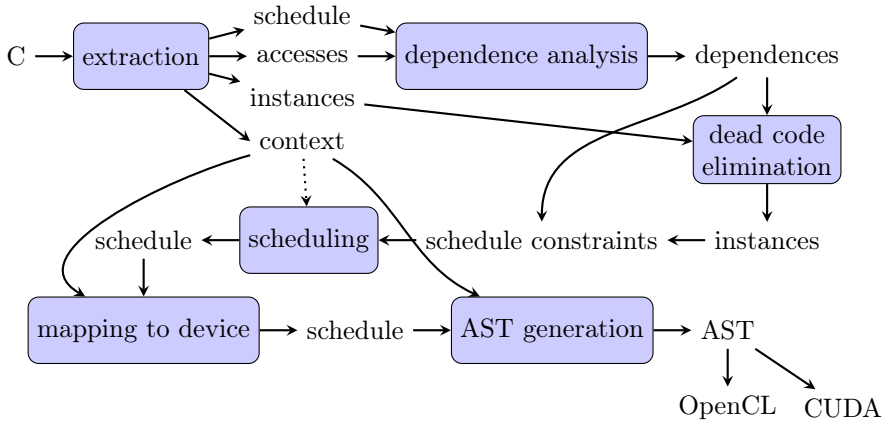

Figure 1: Internal Structure of PPCG

builtins and pragmas. The PENCIL language is described by Baghdadi et al. (2015). More details on how PPCG handles PENCIL are available from Verdoolaege (2015). PPCG also has a mode in which it produces OpenMP code, but this mode is not very advanced yet and the process is somewhat different than that for producing CUDA or OpenCL code. The generation of OpenMP code will not be discussed any further in this report. For the generation of CUDA or OpenCL code, the main tasks of PPCG are

- detecting and exposing the available parallelism,
- mapping parts of the code to an accelerator,
- introducing data copies to and from the accelerator, and
- introducing local data copies on the accelerator.

An overview of the internal structure of PPCG is shown schematically in Figure 1. PPCG first uses pet (Verdoolaege and Grosser 2012) to extract a polyhedral model from the C99/PENCIL code. The main constituents of the extracted model are as follows.

- *instance set*

  The instance set is the set of all "dynamic execution instances", i.e., the set of operations that are performed by the abstracted piece of code. In the case of a model extracted by pet, an operation corresponds to an instance of an expression statement in the source code, where there is a separate instance for each iteration of the enclosing loops. The operations may, however, also correspond to larger pieces of the source code. In particular, if the input contains any dynamic control, then PPCG instructs pet to encapsulate this dynamic control in atomic statements. See Verdoolaege (2015, Section 3.2) for details.

- *access relations*

  An access relation maps elements from the instance set to elements of some data set and expresses which data elements are or may be accessed by a given element of the instance set. pet collects three types of proper access relations, the *may-read access relation*, which collects all the read accesses that may occur in the abstracted fragment; the *may-write access relation*, which collects all the write accesses that may occur in the abstracted

fragment; and the *must-write access relation*, which collects all the write accesses that definitely occur in the abstracted fragment. Clearly, the must-write access relation is a subset of the may-read access relation. Additionally, `pet` also collects a *must-kill access relation* containing pairs of instances and data elements across which no data can flow from a write to a read. The must-kill access relation is explained in more detail in Section 3.2.3 below.

- *schedule*

  The schedule defines a strict partial order, i.e., an irreflexive and transitive relation, on the elements of the instance set that specifies the order in which they are or should be executed. The schedule derived by `pet` corresponds to the original execution order. The representation of schedules is described in Section 4.4 below.

- *context*

  The context holds constraints on some scalar program variables, called *symbolic constants*, that have an unknown but constant value throughout the execution of the analyzed fragment. In particular, values of these symbolic constants for which the analyzed fragment is guaranteed to run into undefined behavior are excluded from the context. Examples of undefined behavior include negative array sizes, division by zero, out-of-bounds accesses and (signed) integer overflow. `pet` also removes values from the context that result in some forms of aliasing, specifically those that result in negative array indices. `pet` also allows users to specify additional constraints by calling `__builtin_assume` or `__pencil_assume`. The single argument to these calls can be any expression, but it will only be added to the context if it expresses a quasi-affine constraint on the symbolic constants. Finally, `PPCG` allows the user to specify additional constraints using the `--ctx` and `--assume-non-negative-parameters` command line options.

For more details about these concepts, see Verdoolaege (2016).

After a polyhedral model has been extracted, `PPCG` uses `isl` to perform dependence analysis. The result of such an analysis is a *dependence relation*, which is a binary relation between elements of the instance set where one of the instances depends on the other in some way. Several types of dependence relations can be considered and the exact nature of the dependence of one instance on the other depends on the type of the dependence relation. Typically, though, the dependence relation expresses that one instance needs to be executed before the other. The dependence analysis used by `PPCG` and the resulting dependence relations are described in Section 3.

The dependences are first used to try and remove some statement instances from the instance set using dead code elimination, which is described by Verdoolaege (2015, Section 4.1). The main purpose of the dependences, however, is to define constraints for the affine scheduler as described in Section 4.3. The result of the scheduler is a new schedule. An overview of the scheduler is presented in Section 4, while details about the scheduling algorithm are explained in Section 6 and Section 7. The schedule is then further modified for mapping onto the accelerator as described in Section 5 and, finally, the schedule is mapped back to an AST using the AST generation of Grosser, Verdoolaege, et al. (2015).

3 Dependence Analysis

3.1 Dependence Analysis in `isl`

The `isl` library contains a generic dependence analysis engine that determines a dependence relation between abstract "sources" and "sinks". This dependence relation contains pairs of source and sink instances for which information may flow from the source to the sink. The input is specified by means of three access relations, each from an abstract instance domain I to an abstract data domain D, and a schedule on I. The output is (essentially) a ternary relation containing triples from I, I and D. In practice, the ternary relation is encoded using wrapped binary relations, mapping a source instance to a pair of sink instance and data element.

The three access relations are the following.

- The *sink access relation* K maps instances to the data elements that they (may) need.
- The *may-source access relation* Y maps instances to the data elements that they may define.
- The *cut access relation* C maps instances to the data elements that cannot pass information across the corresponding instances.

The schedule is used to interpret the terms "before" and "intermediate" below. That is, an instance is considered to come "before" some other instance if it is scheduled before that other instance according to the schedule.

The output *may-dependence relation* maps an instance $\mathbf{i}$ to a pair of instance $\mathbf{k}$ and data element $\mathbf{a}$ if

- Y maps $\mathbf{i}$ to $\mathbf{a}$,
- K maps $\mathbf{k}$ to $\mathbf{a}$,
- $\mathbf{i}$ is scheduled before $\mathbf{k}$,
- there is no intermediate cut of $\mathbf{a}$, i.e., there is no instance between $\mathbf{i}$ and $\mathbf{k}$ that is mapped to $\mathbf{a}$ by C.

As a side product, the dependence analysis also produces a *may-no-source relation* which is a subset of the sink access relation K and which maps an instance $\mathbf{k}$ to $\mathbf{a}$ if there is no cut of $\mathbf{a}$ before $\mathbf{k}$.

This dependence analysis is a generalization of the classical exact dataflow analysis (Feautrier 1991; Pugh and Wonnacott 1994). In particular, exact dataflow is obtained by specifying the read access relation for K and the write access relation for Y and C.

3.2 Dependence Analysis in `PPCG`

`PPCG` uses `isl`'s dependence analysis to compute its dependences, which will in turn be used to constrain the scheduler. There are two modes for computing dependences, one where live-range reordering is enabled (Verdoolaege and Cohen 2016) and one where it is disabled. While live-range reordering is enabled by default, the case where it is disabled is somewhat simpler and will be explained first.

3.2.1 Dependence analysis without live-range reordering

When live-range reordering is disabled, `PPCG` calls the dependence analysis engine three times to compute different dependence relations. In each case, the schedule is set to the original schedule extracted by `pet`.

- flow dependence relation and may-live-in

  The *flow dependence relation* contains pairs of write and read instances such that the value written by the write instance may be read by the read instance. The *may-live-in* relation is a subset of the may-read access relation for which the value being read may have come from outside the analyzed fragment. They are computed by calling the dependence analysis engine with the following input.

  - sink access relation: may-read access relation
  - may-source access relation: may-write access relation
  - cut access relation: union of the must-write access relation and the must-kill access relation

  The resulting may-dependence relation is used as the flow dependence relation after projecting out the accessed element, while the resulting may-no-source relation is used as the may-live-in relation. That is, data may flow from a may-write to a read if there is no intermediate must-write or kill, while all reads for which there is no preceding must-write or kill are considered to be live-in. The use of kills is explained in more detail in Section 3.2.3 below.

- false dependence relation

  The *false dependence relation* contains pairs of accesses where the second access may overwrite data that is used or written by the first access. The input to the dependence analysis engine is as follows.

  - sink access relation: may-write access relation
  - may-source access relation: union of the may-read access relation and the may-write access relation
  - cut access relation: must-write access relation

  The resulting may-dependence relation is used as the false dependence relation after projecting out the accessed element. Note that the cut access relation could also have been left empty. Setting it to the must-write access relation simply removes some false dependences that are (transitively) covered by other false dependences.

- may-live-out

  The *may-live-out* relation is a subset of the may-write access relation for which the value being written may still be needed after the analyzed fragment. It is obtained by removing the writes that are definitely killed from the may-write access relation. These definitely killed writes are obtained by projecting out the sink instance from the may-dependence relation computed with the following input.

  - sink access relation: union of the must-write access relation and the must-kill access relation
  - may-source access relation: may-write access relation

```
A:   a = f1();
B:   f2(a);
C:   a = f3();
D:   f4(a);
```

Listing 2: Live-range reordering example

3.2.2 Live-range reordering

As a trivial example of the need for live-range reordering, consider the code in Listing 2. The flow dependence relation is

$$\{ \mathtt{A}[] \rightarrow \mathtt{B}[]; \mathtt{C}[] \rightarrow \mathtt{D}[] \}, \tag{1}$$

while the false dependence relation is

$$\{ \mathtt{A}[] \rightarrow \mathtt{C}[]; \mathtt{B}[] \rightarrow \mathtt{C}[] \}. \tag{2}$$

As will be explained in more detail in Section 4.3, both flow dependences and false dependences need to be respected by the schedule. In particular, the source of each such dependence needs to be scheduled before the sink of the dependence. In the example, this means that the order of execution is completely fixed by the dependences. However, it should be clear that the two live-ranges of `a`, one from `A` to `B` and one from `C` to `D`, can be interchanged without affecting the correctness of the program. Live-range reordering (Verdoolaege and Cohen 2016) allows such live-ranges to be reordered by using a slightly different classification of dependences and instructing the scheduler to take them into account. In particular, during the recursive construction of the schedule, constraints keeping live-ranges apart are only taken into account at those levels of the schedule where the end-points of the live-ranges are not scheduled together. Live-ranges that have there end-points scheduled together at some level are called *local* to that level. In the example, if the outer level of the schedule schedules `A` and `B` together and also schedules `C` and `D` together, then the constraints relating the two live-range do not need to be taken into account at that level and the two live-ranges can be freely reordered. Assuming the outer level schedules the two live-ranges apart, the next level only needs to schedule `B` after `A` and `D` after `C`. The relevant dependence relations are described in Section 3.2.5, while their mapping to schedule constraints is described in Section 4.3.

3.2.3 Kills

A must-write definitely overwrites any data that was written by a preceding write and therefore ensures that no data can flow from such a preceding write to a read that occurs after the must-write. The must-write is said to kill the dependence between the preceding write and the following read. In some cases, it may be useful for the user to manually introduce additional kills. The `__pencil_kill` builtin can be called for this purpose (Verdoolaege 2015, Section 3.8).

Example 1 *As a simple example, consider the code shown in Listing 3 on the following page. Without the `__pencil_kill` call at the end of the analyzed fragment, which is delimited by `#pragma scop` and `#pragma endscop`, the compiler would need to assume that `t` may be used after the loop. In particular, this means that the last write needs to remain last. This limits the scheduling freedom as the iterations of the loops cannot be freely reordered, even with live-range*

```
void f(int n, int A[restrict static n],
        int B[restrict static n])
{
        int t;
#pragma scop
        for (int i = 0; i < n; ++i) {
                t = A[i];
                B[i] = t;
        }
        __pencil_kill(t);
#pragma endscop
}
```

Listing 3: Kill example

reordering enabled. The call to `__pencil_kill` *expresses the fact that no data can flow through* `t` *across that statement, i.e., that the data in* `t` *is not needed after the call. As a result of adding this kill to the sink access relation during the may-live-out computation, the last write to* `t` *is removed from the may-live-out relation, thereby also removing the requirement to keep this write last.*

Besides kills that are manually added through calls to `__pencil_kill`, `pet` also collects some kills automatically. In particular, any variable that is declared inside the analyzed fragment is killed both at the point of the declaration and at the end of the enclosing block (provided this enclosing block is part of the analyzed fragment). Furthermore, any variable declared inside the scope that contains the analyzed fragment and that is not used after the analyzed fragment is killed at the end of the analyzed fragment.

Example 2 *Continuing from Example 1 on the previous page, the variable* `t` *in Listing 3 is declared inside the scope that contains the analyzed fragment and it is not used after this fragment. The explicit call to* `__pencil_kill` *is therefore not strictly needed since the kill is inserted automatically by* `pet`*.*

3.2.4 Tagged access relations and dependence relations

While the standard flow dependence relation computed above, which maps instances to instances, is sufficient for constraining the scheduler in case of no live-range reordering (as explained in Section 4.3 below), additional steps in the mapping to an accelerator may require more refined information (see Section 5.1 and Section 5.2). In particular, some information is needed about which data elements are involved in the dependence. A statement instance in itself does not provide enough information because any given statement may read and/or write many different data elements, especially if the statement corresponds to a compound statement that encapsulates dynamic control. The statement instance is therefore augmented with additional information in both the access relations and the dependence relations. This process is called *tagging* and results in *tagged access relations* and *tagged dependence relations*.

One option for tagging would be to introduce the accessed data element as the tag, but it is usually sufficient to only consider the reference through which the data elements are accessed. The latter choice should be more efficient because it does not increase the total dimension of the relations involved and it is also the choice taken by `PPCG`. Note that in general this does in fact lose some information since a given array reference may access several elements. This

happens in particular when the reference appears in a call to a function and if this function (or its summary function, see Baghdadi et al. 2015; Verdoolaege 2015) accesses more than one element in the array or more than one element in a structure. If the body of the function (or its summary function) is not available then all elements accessible through the reference are assumed to be accessed.

The tags for the references are generated by `pet` and `PPCG` includes them in the access relations that are used to compute the flow dependence relation, or rather the *tagged flow dependence relation*. Note that since the domains of the access relations need to be the same as that of the schedule in the input of the dependence analysis engine of Section 3.1, the tags also need to be introduced in the schedule. The resulting may-dependence relation and may-no-source relation then also include those tags.

3.2.5 Dependence analysis with live-range reordering

When live-range reordering is enabled, `PPCG` calls the dependence analysis engine seven times. The computation of the tagged flow dependence relation, the may-live-in relation, the false dependence relation and the may-live-out relation is the same as in the case of no live-range reordering of Section 3.2.1 using three calls. Note that as explained in Section 4.3 below, the false dependence relation is only used for historical reasons. The remaining four calls are used to compute variations of the false dependences that are needed for live-range reordering. These false dependences come in two classes, the forced dependences and the order dependences. The *forced dependence relation* contains dependences that need to be respected irrespective of any live-range reordering. The *tagged order dependence relation* contains dependences that only need to be respected in case there is a risk that some live-ranges might overlap, i.e., when at least one of the adjacent live-ranges is not local. This will be explained in more detail in Section 4.3. As in Section 3.2.1, the schedule used during the computation of these dependence relations is the original schedule extracted by `pet`, with tags added as in Section 3.2.4 if a tagged dependence relation is being computed.

- tagged order dependence relation

  The inputs to the dependence analysis engine are as follows.

  - sink access relation: tagged may-write access relation
  - may-source access relation: union of the tagged may-read access relation and the *unmatched* tagged may-write access relation

  The order dependences are used to prevent live-ranges that are not local to that level from overlapping by imposing their original order. These dependences therefore relate the reads of live-ranges with the writes of subsequent live-ranges. Unlike the case of the computation of the false dependence relation in Section 3.2.1, the cut access relation is left empty because any write that would be used to cut dependences may have been reordered away from in between the two live-ranges and would therefore not be able to prevent those two live-ranges from overlapping. The unmatched tagged may-write access relation is the subset of the tagged may-write access relation that has domain elements that do *not* appear in the domain of the tagged flow access relation. These lone writes essentially form degenerate live-ranges that are not otherwise taken into account. Note that dead code elimination can usually eliminate statements containing such lone writes, but it is not possible to eliminate all such statements in general.

- forced dependence relation

  The forced dependence relation consists of three parts. The first part ensures that live-out writes are not overwritten. It is computed with the following inputs.

  - sink access relation: may-live-out relation
  - may-source access relation: may-write access relation

  The second part ensures that data read by live-in reads is not overwritten. It is computed with the following inputs.

  - sink access relation: may-write access relation
  - may-source access relation: may-live-in relation

  The final part ensures that live-range sources that share the same sink and that access the same memory element are executed in their original order. This part is computed with the following inputs.

  - sink access relation: a relation mapping the sources of the tagged flow access relation to pairs consisting of the corresponding sinks and the corresponding elements in the tagged may-write access relation. The tags are removed from the sources of this relation.
  - may-source access relation: the same relation

4 Scheduler Interface

This section describes the interface of the scheduler, i.e., the input and the output, as well as the way this input is constructed from the dependence relations by `PPCG`.

4.1 Optimization Criteria

This section briefly describes the scheduler optimization criteria that are of interest to `PPCG`. These mainly have an effect on the choice of scheduling algorithms, as discussed below in Section 6, but they also affect the way the input to the schedule is specified.

Some of the main requirements for `PPCG` are the following.

- Correctness

  Clearly, the scheduler should produce a schedule that preserves the semantics of the original code. That is, the generated schedule should exhibit *validity*.

- Two levels of parallelism

  An accelerator typically offers two levels of parallelism. In CUDA-speak, these are called blocks and threads. In order to be able to successfully exploit the architecture, two levels of parallelism should then also be exposed in the application. This highlights the second criterion for the scheduler: detection/exposure of *parallelism*. In `PPCG`, the second level of parallelism is obtained through tiling, leading to the third criterion: *tilability*.

- Reduced working set

  It can be advantageous to have reduced working sets for some arrays such that they can be mapped to shared memory (CUDA-speak) or registers. In `PPCG`, this reduced working set is obtained through the same tiling that ensures a second level of parallelism, reinforcing the *tilability* criterion.

- Reduced data movement

  Data should be used as close as possible to where it is produced to reduce the need for data movement. That is, the constructed schedule should exhibit *locality*.

- Simple schedules

  Finally, the generated schedule should be as simple as possible since it will be used in several subsequent steps, including the AST generation at the very end. In particular, the coefficients should be as small as possible. That is, the scheduler should strive for *simplicity*.

4.2 Schedule Constraints

This section describes the constraints that can be imposed on the `isl` scheduler in order to be able to meet optimization criteria such as those mentioned in Section 4.1. There are several types of schedule constraints, each consisting of pairs of statement instances. The constraint imposed on the schedule by these pairs is explained in terms of an integer-valued affine schedule function f. As described in Section 4.4 below, the actual schedule produced by the scheduler takes the form of a tree with these affine functions as core elements.

- *validity schedule constraint*

  A validity constraint $\mathbf{a} \to \mathbf{b}$ imposes that instance $\mathbf{b}$ is scheduled after instance $\mathbf{a}$, i.e.,

  $$f(\mathbf{b}) \geq f(\mathbf{a}), \tag{3}$$

  with f the schedule function. These constraints can be used to ensure validity of the schedule.

- *proximity schedule constraint*

  A proximity constraint $\mathbf{a} \to \mathbf{b}$ specifies that instance $\mathbf{b}$ should preferably be executed close to $\mathbf{a}$, i.e., the distance

  $$f(\mathbf{b}) - f(\mathbf{a}) \tag{4}$$

  should be as small as possible (in absolute value). These constraints can be used to favor locality of the schedule.

- *coincidence schedule constraint*

  A coincidence constraint $\mathbf{a} \to \mathbf{b}$ specifies that instance $\mathbf{b}$ should preferably be executed together with $\mathbf{a}$, i.e.,

  $$f(\mathbf{b}) = f(\mathbf{a}), \tag{5}$$

  for as long as possible, i.e., for as many of the outer levels of the schedule as possible. As explained in Section 4.4 below, the successful application of the coincidence constraints is also marked explicitly in the resulting schedule. These constraints can be used to favor parallelism in the schedule.

- *conditional validity schedule constraint*

  A conditional validity constraint consist of two parts, a condition $\mathbf{b} \to \mathbf{c}$ and a conditioned validity constraint $\mathbf{a} \to \mathbf{b}$ or $\mathbf{c} \to \mathbf{d}$. A conditioned validity constraint is only imposed if there is an adjacent condition that "holds", where a conditioned validity constraint is *adjacent* to a condition if the start point of one is the end point of the other and a condition

$\mathbf{b} \to \mathbf{c}$ "holds" if $\mathbf{b}$ and $\mathbf{c}$ are *not* coscheduled. The conditional validity schedule constraint may optionally be tagged in the same way that access relations and dependence relations are tagged in Section 3.2.4. In this case, the tags are taken into account while determining adjacency, but they are projected out for determining whether a condition is local and for imposing a conditioned validity constraint. Conditional validity schedule constraints are mainly useful for live-range reordering, as explained in Section 4.3 below.

Note that a schedule constraint that has its end-points scheduled apart at some level of the schedule is said to be *carried* at that level. Carried schedule constraints are no longer taken into account at deeper levels of the schedule. That is, only schedule constraints that are coscheduled by all outer levels are taken into account.

4.3 Dependences and Schedule Constraints

In case of traditional dependence relations (i.e., without live-range reordering), the dependence relations can be used to set schedule constraints as follows.

- flow dependence relation

  The flow dependence relation needs to be added to the validity schedule constraints to ensure that reads are scheduled after the corresponding writes. They are typically also added to the proximity schedule constraints such that the reads would be scheduled closely after the corresponding writes. If parallelism is an optimization criterion, then they should also be added to the coincidence schedule constraints.

- false dependence relation

  The false dependence relation needs to be added to the validity schedule constraints to ensure that write instances that may overwrite data needed or written by some other instance are scheduled after these other instances. If parallelism is an optimization criterion, then they should also be added to the coincidence schedule constraints. They can optionally also be added to the proximity schedule constraints if it is important for reuses of the same memory element to be scheduled close together. At present, `PPCG` does in fact add the false dependence relation to the proximity schedule constraints, but this is mainly for historical reasons. The effect of this choice in `PPCG` has not been evaluated in detail.

- "input dependence relation"

  It may also be useful to consider pairs of read instances that have the same corresponding write. Such pairs are sometimes called "input dependences". For writes that occur outside the analyzed fragment, the corresponding reads are pairs of live-in reads that read from the same data element. These pairs can be added to the proximity schedule constraints to encourage reuses of the same value to be scheduled close together. `PPCG` does not currently consider such input dependences.

When live-range reordering is enabled, `PPCG` uses the dependence relations computed in Section 3.2.5 to set schedule constraints in the following way.

- validity schedule constraints

  The validity schedule constraints are composed of the flow dependence relation and the forced dependence relation.

- proximity schedule constraints

  The proximity schedule constraints are composed of the flow dependence relation and the false dependence relation. Again, the false dependence relation is added to the proximity schedule constraints for mainly historical reasons.

- coincidence schedule constraints

  The coincidence schedule constraints are composed of the flow dependence relation, the forced dependence relation and the subset of the order dependence relation that is derived from accesses to proper arrays, i.e., not scalars. This last piece is needed because executing live-ranges that access the same memory elements in parallel requires some form of array expansion (Feautrier 1988a) and `PPCG` currently only supports this expansion for scalars.

- conditional validity schedule constraints

  The conditions are set to the tagged flow dependence relation, while the conditioned validity constraints are set to the tagged order dependence relation. This ensures that if any live-range is not local at a certain level, then all the adjacent order dependences are imposed, ensuring the live-range does not overlap with any other live-ranges accessing the same memory element.

4.4 Schedule Trees

The output of the scheduler is a schedule in the form of a tree (Verdoolaege, Guelton, et al. 2014). The relative order of two instances is determined by the outermost node in the schedule tree that schedules the instances apart. The basic construct in a schedule tree is an integer-valued affine function that prescribes the execution order based on increasing function values. These affine affine functions may be combined into bands, which may in turn be combined into a tree along with several other types of nodes. For the purpose of this report, the main node types are the following.

- *band*

  A band node specifies a relative order on the instances according to the associated multi-dimensional piecewise quasi-affine partial schedule. That is, if the associated function assigns different sequences of values to two instances, then the first position with a different value determines the order of the two instances. If the two instances are assigned the same sequence of values then they are coscheduled by the band node. The piecewise quasi-affine functions that form the multi-dimensional piecewise quasi-affine function are called its *members*. If the members can be freely reordered without affecting the validity of the schedule, then the band is marked *permutable*. A member that coschedules all pairs of instances in the coincidence schedule constraints that have not been carried by outer nodes in the tree, is marked *coincident*.

- *sequence*

  A sequence node partitions the instances through child *filter* nodes that are executed in the given order.

- *set*

  A set node partitions the instances through child *filter* nodes that may be executed in any order.

```
for (int i = 0; i < M; i += 1)
  for (int j = 0; j < N; j += 1) {
S1: C[i][j] = 0;
    for (int k = 0; k < K; k += 1)
S2:   C[i][j] = (C[i][j] + (A[i][k] * B[k][j]));
  }
```

Listing 4: Matrix multiplication

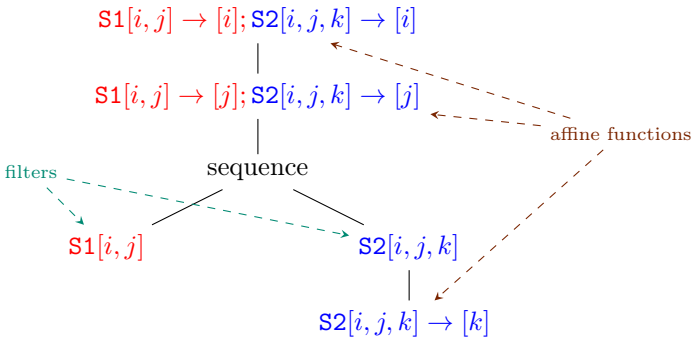

Figure 5: Schedule tree for the code in Listing 4

- *leaf*

  A leaf node is an explicit marker that is used in the leaves of the tree.

- *expansion*

  An expansion node expands each instance that reaches the node to one or more instances. This allows groups of instances (the expanded instances) to be treated as a single instance in the part of the tree above the expansion node.

Note that the core `isl` scheduler will not itself produce expansion nodes.

Example 3 *A schedule tree that corresponds to the original execution order can easily be derived by creating a single-dimensional band for each loop and a sequence for each compound statement, with a child filter node for each statement in the compound statement. For the code shown in Listing 4, this results in the tree shown in Figure 5. The outer two nodes of the tree correspond to the outer two loops in the code. In each case, the instances are executed in the order of increasing values of the corresponding loop iterator. The next node is a sequence corresponding to the compound statement, with children for the two statements that form the compound statement. The first child does not have any further descendants because the outer nodes already determine the execution order of the `S1`-instances completely. The second child, on the other hand, still has a further band node corresponding to the innermost loop.*

5 Device Mapping

The mapping to the device starts out from the schedule tree produced by the `isl` scheduler (or the original schedule if `PPCG` has been instructed to preserve this original schedule using the `--no-reschedule` command line option). If this

schedule tree does not contain any coincident band member, then there is clearly no point in using the accelerator and pure CPU code is generated instead.

Otherwise, the following operations are performed.

1. select subtree for mapping onto the device

   The selected subtree is currently the entire schedule tree, except that the following nodes are excluded.

   - coincidence-free children of outermost set node

     The children of the set node can be freely reordered. This means in particular that some of them can be executed on the CPU, while others are executed on the device. There is no point in executing the children that have no coincident band members on the device, so they are executed on the CPU instead.

   - coincidence-free initial children of outermost sequence node

     As before, there is no point in executing these children on the device. Instead, they are executed on the CPU prior to the initialization of the device. Any data that is produced by these children and that is needed by the children that are mapped to the device is copied to the device in the same way that live-in data is copied to the device.

2. generate kernels

   Within the selected tree, kernels are generated for the outermost permutable bands with coincident members, as well as for all maximal subtrees that are free of bands with coincident members. In practice, a zero-dimensional band node is inserted before these maximal coincidence-free subtrees and these zero-dimensional band nodes are then treated in the same way as the outermost permutable bands with coincident members. The construction of a kernel from such a band node will be described in Section 5.1.

3. add data copying to/from device around selected subtree

   This operation will be described in Section 5.2.

4. add device initialization and clean-up around entire schedule tree

   The initialization consists of declaring device arrays, initializing the device and allocating the device arrays. The device arrays correspond to the arrays that are accessed by code mapped to the device and that are stored in global memory. In particular, no device arrays are created for arrays that are completely mapped to registers or shared memory. This mapping to registers or shared memory will be described in Section 5.1.

5.1 Kernel Generation

Given a band node with at least one coincident member (or a zero-dimensional band node that was inserted on top of a coincidence-free subtree), the band is first tiled, resulting in an outer *tile band* and an inner *point band*. The coincident members of the tile band are mapped to CUDA blocks, while the coincident members of the point band are mapped to CUDA threads. If there are more than two (or three) coincident members, then only the outermost two are mapped to blocks and the outermost three are mapped to threads. The motivation for the tiling is that it introduces an extra level of parallelism and that it typically reduces the working sets of arrays within the point band. The reduced working sets result in more opportunities for mapping arrays to CUDA shared memory or registers.

```
for (int c0 = 0; c0 < M; c0 += 1)
  for (int c1 = 0; c1 < N; c1 += 1) {
    C[c0][c1] = 0;
    for (int c2 = 0; c2 < K; c2 += 1)
      C[c0][c1] = (C[c0][c1] + (A[c0][c2] * B[c2][c1]));
  }
```

Listing 6: Matrix multiplication before tiling

```
for (int c0 = 0; c0 < M; c0 += 32)
  for (int c1 = 0; c1 < N; c1 += 32)
    for (int c2 = 0; c2 < K; c2 += 32)
      for (int c3 = 0; c3 <= ppcg_min(31, M - c0 - 1); c3 += 1)
        for (int c4 = 0; c4 <= ppcg_min(31, N - c1 - 1); c4 += 1) {
          if (c2 == 0)
            C[c0 + c3][c1 + c4] = 0;
          for (int c5 = 0; c5 <= ppcg_min(31, K - c2 - 1); c5 += 1)
            C[c0 + c3][c1 + c4] = (C[c0 + c3][c1 + c4] +
                        (A[c0 + c3][c2 + c5] * B[c2 + c5][c1 + c4]));
        }
```

Listing 7: Matrix multiplication after tiling

Example 4 *As an example, take the matrix multiplication code in Listing 6. This is the same code as that shown in Listing 4 on page 14, but now it is considered to have been generated from a schedule tree consisting of a band node with three members, the first two of which are marked coincident. After tiling this band, the generated code looks as in Listing 7, where the boxed part of the code corresponds to the point band. Within this point band, only a single element of* `C` *is used per (virtual) thread, while a fixed-size tile of* `A` *and* `B` *is used. This means that* `C` *can be mapped to a register, while* `A` *and* `B` *can be mapped to shared memory.*

PPCG does not currently use any sort of performance model for determining appropriate tile, grid and block sizes. This means that these sizes should be specified by the user using the `--sizes` command line option. If the sizes are not explicitly specified, then some fixed defaults are used that may not always be very meaningful. Note that the band structure depends on the output of the scheduler, which cannot in general be predicted by the user. The user therefore typically needs to run PPCG twice, once to see which kernels get generated and once to set the sizes.

Returning to the mapping to registers or shared memory, PPCG does not perform this mapping on arrays as a whole, but rather on groups of array references. This allows multiple copies to be created for different references to the same array. Note, though, that if any data is written to the array, then the corresponding element should only appear in a single copy to ensure consistency. This is the basis for the construction of the array reference groups. In particular, a separate group is initially created for each individual array reference. These groups are then incrementally combined as long as there is a pair of conflicting groups, where groups are conflicting if they may access the same data and if at least one of the groups performs a write.

Example 5 *Consider for example the code fragment in Listing 8 on the next page. There are two references to* `A`*, but they both perform a read, so the two references do not need to be combined into a single reference group. Only the*

```
for (i = 0; i < _PB_N; i++) {
  for (j = 0; j <= i; j++)
    C[i][j] *= beta;
  for (k = 0; k < _PB_M; k++) {
    for (j = 0; j <= i; j++)
      C[i][j] += alpha * A[i][k] * A[j][k];
  }
}
```

Listing 8: Excerpt from `PolyBench/C 4.1` `syrk` benchmark

first reference is mapped to shared memory by `PPCG`*. The other reference is left to read data from global memory.*

The choice of whether to use shared memory or registers is based on the criteria described below. First of all, no copies are created for the following arrays references.

- accesses to read-only scalars

  Read-only scalars are not stored in global memory in the first place, but are instead passed as function arguments to the kernel.

- references that reference more than one element of an array or structure

  Since `PPCG` can only modify the reference, the layout of the referenced slice of the array would have to be the same in shared memory and this is not currently handled by `PPCG`.

- any may-writes that are not also must-writes

  Since the data is not guaranteed to be written, the data stored in global memory prior to the potential write would have to be copied to shared memory or registers, such that it would not get overwritten by the write-out from shared memory or registers to global memory. This is currently not supported by `PPCG`, but it would be similar to the handling of copies to/from the device in Section 5.2 below.

Second, a copy to shared memory or registers is only created if the accessed set fits within a rectangular tile with fixed bounds. If, moreover, the access exhibits some reuse or if it is uncoalesced, then a copy in shared memory may be created. If the access exhibits some reuse, every element is accessed by a single thread and the index expressions only depend on coincident band members, then a copy in registers may be created. The condition on index expressions is needed to ensure that the band can be sunk to the leaves of the tree and subsequently unrolled. This in turn is needed because registers are not addressable in CUDA. If both options are available then the choice depends on the position where data copy operations are inserted, as described next.

If any mapping is created, then explicit data copy operations, along with synchronization to protect the local copies (Verdoolaege 2015, Section 4.6), need to be inserted in the schedule tree. By default, they are inserted right outside the band that is mapped to threads, but they can be moved up if this movement does not affect the tile that is mapped to registers or shared memory. If an array reference group can potentially be mapped to both shared memory and registers then the choice that allows the copies to be inserted at the highest location in the tree is taken. If the two positions are the same, then registers are preferred. Note that data copy operations to/from global memory are only inserted if some

```
for (int c0 = 32 * b0; c0 < M; c0 += 8192)
  for (int c1 = 32 * b1; c1 < N; c1 += 8192) {
    for (int c2 = 0; c2 < K; c2 += 32) {
      if (M >= t0 + c0 + 1)
        for (int c4 = t1; c4 <= ppcg_min(31, K - c2 - 1); c4 += 16)
          shared_A[t0][c4] = A[(t0 + c0) * K + (c2 + c4)];
      if (K >= t0 + c2 + 1)
        for (int c4 = t1; c4 <= ppcg_min(31, N - c1 - 1); c4 += 16)
          shared_B[t0][c4] = B[(t0 + c2) * N + (c1 + c4)];
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
      if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1 && c2 == 0) {
        private_C[0][0] = 0;
        if (N >= t1 + c1 + 17)
          private_C[0][1] = 0;
      }
      if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1)
        for (int c3 = 0; c3 <= ppcg_min(31, K - c2 - 1); c3 += 1) {
          private_C[0][0] = (private_C[0][0] +
                        (shared_A[t0][c3] * shared_B[c3][t1]));
          if (N >= t1 + c1 + 17)
            private_C[0][1] = (private_C[0][1] +
                        (shared_A[t0][c3] * shared_B[c3][t1 + 16]));
        }
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    }
    if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1) {
      C[(t0 + c0) * N + (t1 + c1)] = private_C[0][0];
      if (N >= t1 + c1 + 17)
        C[(t0 + c0) * N + (t1 + c1 + 16)] = private_C[0][1];
    }
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
  }
```

Listing 9: Matrix multiplication mapped to device

data may flow in/out. This test is based on the tagged flow dependence relation. Furthermore, the data copy from global memory to shared memory performs a copy of the entire tile rather than of the exact piece of the array that is accessed. Copying the entire tile usually results in simpler code, which in turn leads to a reduced risk of thread divergence.

Example 6 *As an example of the data copy insertion, consider the code shown in Listing 9. The* `c2`*-loop corresponds to the innermost band member of the tile loop. The copies to shared memory are therefore initially created inside this loop. While the copying from global memory to the shared memory copies of* `A` *and* `B` *remains in this position, the copying from the register copy of* `C` *to global memory has been hoisted out of this loop. Note also that no data is copied back from the shared memory copies of* `A` *and* `B` *or from global memory to the register copy of* `C` *because there is no data-flow out of* `A` *and* `B` *or into* `C`*.*

5.2 Data Copying to/from Device

Recall that kernels are created for each outermost permutable band within a selected subtree of the schedule tree returned by the `isl` scheduler and that data copying to/from the device is inserted around this selected subtree. The analysis of which data to copy is similar to the approximation formulation of Alias et al. (2011). The description below is formulated in terms of data sets, but in principle the selected subtree may have outer band nodes and the data sets may depend on the iterators corresponding to these band nodes. The actual implementation in `PPCG` therefore uses binary relations that map these iterators to data sets.

```
__pencil_kill(A);
for (int i = 0; i < n; i++)
    if (B[i] > 0)
        A[i] = B[i];
```

Listing 10: Data copying example

The algorithm first determines the *copy-out* set, i.e., the data that must be copied back from the device to the CPU. In particular, the set of all may-writes is taken and those writes that are only needed for dataflow within the selected subtree are removed. This computation is based on the tagged flow dependence relation. Finally, the copy-out set is approximated to entire arrays. That is, a complete array is copied out as soon as one of its elements needs to be copied out.

Then the *may-persist* set is computed. This is the set of elements that may need to be preserved by the selected subtree. This set consists of the elements that may need to be preserved by the entire analyzed fragment, i.e., those elements that are not definitely written and not definitely killed, along with the elements that are in potential dataflow across the selected subtree.

The *may-not-written* set is then the set of (relevant) elements that may be copied out without being written first. It is computed by first intersecting the copy-out set with the may-persist set and then removing those elements that are definitely written by the selected subtree.

Finally, the *copy-in* set is the set of elements that need to be copied from the CPU to the device prior to the execution of the kernels. This set is the union of the live-in elements and the may-not-written set.

Note that if the array elements are structures, then entire structures are copied in/out. That is, no attempt is currently made by `PPCG` to only copy fields that are effectively being accessed.

Example 7 *As an example, consider the code fragment in Listing 10. The* `A`*-array may be written by the code fragment and so it is included in the copy-out set. However, some elements may also end up* not *being written by the fragment. Without the call to* `__pencil_kill`*,* `PPCG` *would have to assume that parts of* `A` *may be expected to survive across the fragment such that* `A` *would also be added to the copy-in set. Through the call to* `__pencil_kill`*, the user expresses that* `A` *is not expected to be preserved, avoiding the need to copy* `A` *to the device.*

6 Core Schedule Algorithms

This section describes the scheduler implemented in `isl` and used by `PPCG`. It takes schedule constraints derived from dependences (as described in Section 4.3) as input and produces a new schedule from scratch. `isl` also has some support for modifying a schedule and, possibly with some further extensions, it would be possible to use the same optimization criteria described in this section to modify parts of the original schedule. However, the current version of `PPCG` will simply call the `isl` scheduler to construct a new schedule, unless `PPCG` is instructed to keep the original schedule.

6.1 Components

Before the core `isl` scheduler is called to construct a new band node, the weakly connected components of the *statement-level schedule constraint graph* are first

computed, where the statement-level schedule constraint graph is the graph that has a node for each statement and an edge between two nodes if there is any schedule constraint between instances of the corresponding statements. If more than one weakly connected component is detected, then a set node is introduced, with a child for each component, and the core scheduler is called on each component separately.

If the `--schedule-serialize-sccs` option is set, then *strongly* connected components of the statement-level schedule constraint graph are computed, but only the validity schedule constraints and the conditional validity schedule constraints are taken into account during the construction of the strongly connected components. If more than one strongly connected component is detected, then a sequence node is introduced, with a child for each component in topological order, and the core scheduler is called on each component separately.

6.2 Forced Outer Coincidence

Recall from Section 4.1 that aside from validity, the main optimization criteria for `PPCG` are parallelism, tilability, locality and simplicity. Several scheduling algorithms have been proposed within the context of polyhedral compilation and the `isl` scheduler uses a combination of two of these scheduling algorithms in an attempt to meet these criteria.

- the Feautrier scheduler (Feautrier 1992b)

  The Feautrier scheduler tries to carry as many (validity) schedule constraints as possible at each level until no constraints are left to carry. The statement instances that are coscheduled by the constructed schedule can be executed in parallel. This results in fine-grainded parallelism. The Feautrier scheduler will be described in more detail in Section 6.4.

- the Pluto scheduler (Bondhugula, Baskaran, et al. 2008)

  The Pluto scheduler has two main objectives. It tries to improve locality by minimizing the distances over dependences, with coarse-grainded parallelism as an extreme case where the distances are zero. Furthermore, it tries to create a sequence of independent affine scheduling functions satisfying the same constraints, i.e., without removing the constraints carried by earlier affine scheduling functions. This results in a sequence of permutable scheduling functions that therefore form a tilable band.

The scheduling algorithm implemented in `isl` and used by `PPCG` is a variation of the Pluto scheduler that uses the Feautrier scheduler as a fallback when this variation fails to produce a solution. In particular, parallelism is crucially important for mapping code to an accelerator. The scheduler therefore *forces* each band to have *outer* coincident members, which also explains the choice for the Pluto scheduler as the main scheduling algorithm. Note that since the members in the band are permutable, if the outermost member cannot be made coincident, then none of the members can be coincident. If having coincident members proves to be impossible then a single step of the Feautrier scheduler is used to remove as many (validity) schedule constraints as possible. At this point, the scheduler has fewer constraints to take into account and the construction of a permutable band with coincident members is attempted once more. This process continues until a complete schedule has been constructed. Note that forcing coincident members is an optional feature of the `isl` scheduler that is turned on by `PPCG` by setting the `--schedule-outer-coincidence` option.

An alternative to this combinations that tries to force outer coincident members would be to first apply the regular Pluto scheduler and then to apply a

```
  for (t = 0; t < _PB_TSTEPS; t++) {
      for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
S:        B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                       A[i][1+j] + A[1+i][j] + A[i-1][j]);
      for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
T:        A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                       B[i][1+j] + B[1+i][j] + B[i-1][j]);
    }
```

Listing 11: Excerpt from `PolyBench/C 4.1` `jacobi-2d` benchmark, with additional statement labels

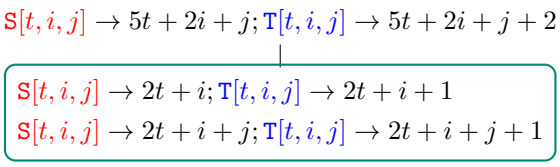

Figure 12: Result of applying the Pluto scheduler + wavefront to the code fragment in Listing 11

wavefront transformation on the bands that have no coincident members. For this purpose, assume that the coincidence schedule constraints form a subset of the validity schedule constraints. Let f_i be the n members of the band. Since the band is part of a valid schedule, $f_i(\mathbf{b}) \geq f_i(\mathbf{a})$ holds for each validity schedule constraint $\mathbf{a} \to \mathbf{b}$ and for each i. This means that $f(\mathbf{b}) \geq f(\mathbf{a})$ also holds for the sum $f = \sum_i f_i$, i.e., f is a valid schedule at this point in the schedule tree. Furthermore, any schedule constraint carried by at least one of the f_i is also carried by f. The original band can therefore be replaced by a sequence of two bands,

1. an outer band with a single member f, and
2. an inner coincident band with $n - 1$ out of the n original band members.

That is, every coincidence schedule constraint that would prevent any of these $n - 1$ members from being coincident is carried by the outer band and therefore no longer affects the inner band. The main drawback of this alternative compared to the combination with the Feautrier scheduler fallback that `PPCG` ended up using is that the sum f may have relatively large coefficients, making it score less favorably on simplicity. Note that Lim and Lam (1997, Section 7.3) perform a similar wavefront transformation on their maximally independent partition mappings.

Example 8 *As an example of the difference between the combination used by* `PPCG` *and the application of a wavefront on the output of the Pluto scheduler, consider the code fragment shown in Listing 11. The Pluto scheduler produces the following single-band schedule:*

$$\begin{aligned} &\mathtt{S}[t,i,j] \to t; \mathtt{T}[t,i,j] \to t \\ &\mathtt{S}[t,i,j] \to 2t+i; \mathtt{T}[t,i,j] \to 2t+i+1 \\ &\mathtt{S}[t,i,j] \to 2t+i+j; \mathtt{T}[t,i,j] \to 2t+i+j+1. \end{aligned} \tag{6}$$

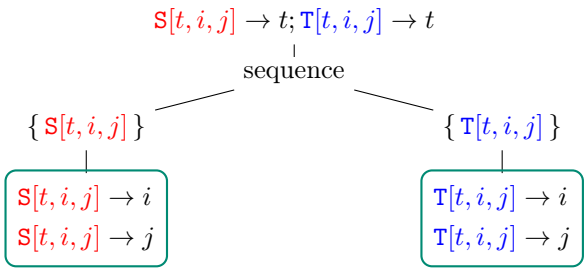

Figure 13: Result of applying a combination of the Pluto scheduler and the Feautrier scheduler to the code fragment in Listing 11 on the previous page

The sum of the affine schedule functions is

$$\mathtt{S}[t,i,j] \to 5t+2i+j; \mathtt{T}[t,i,j] \to 5t+2i+j+2. \tag{7}$$

The resulting schedule tree is therefore as shown in Figure 12 on the previous page. When using the combination scheduler of `isl` *on the other hand, the first application of the Pluto scheduler fails to produce a solution because no coincident members can be found. An application of the Feautrier scheduler produces the band shown at the top of Figure 13. Actually, an application of the standard Feautrier scheduler would produce the schedule function*

$$\mathtt{S}[t,i,j] \to 2t; \mathtt{T}[t,i,j] \to 2t+1 \tag{8}$$

instead, but this is transformed into the band shown in Figure 13 using the technique described below in Section 7.1. After removing all carried schedule constraints, the schedule constraint graph can be decomposed and a sequence node is introduced that schedules all instances $\mathtt{S}[t,i,j]$ *before all instances* $\mathtt{T}[t,i,j]$*. Within each child, all schedule constraints have been removed and a fully coincident band can be constructed. (In practice, a schedule band is first computed for each component separately and then it is determined that the two components should not be combined. This is described in Section 7.3 below.) In this case, the constructed schedule corresponds to the original execution order. Note that the coefficients in the schedule in Figure 13 are smaller than those in the schedule in Figure 12 on the previous page. This would still be the case even without the optimization of Section 7.1. The larger coefficients of Figure 12 on the preceding page can have a dramatic effect because of the accumulation in the subsequent steps of Section 5.*

6.3 The Farkas Lemma

Both the Feautrier scheduler and the Pluto scheduler use the same mechanism for converting the schedule constraints of Section 4.2 to constraints on the schedule coefficients. Recall that schedule constraints are given in terms (pairs of) statement instances, while the affine functions $f(\mathbf{x})$ of which the schedule is composed are affine functions in precisely these statement instances $\mathbf{x}$ and are described by the schedule coefficients $\mathbf{c}^{\mathrm{x}}$, $\mathbf{c}^{\mathrm{n}}$ and c^{c} in

$$f(\mathbf{x}) = \mathbf{c}^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}^{\mathrm{n}} \cdot \mathbf{n} + c^{\mathrm{c}}, \tag{9}$$

with $\mathbf{n}$ the symbolic constants. In particular, both schedulers apply the *Farkas lemma* (Schrijver 1986, Corollary 7.1h, page 93; Feautrier 1992a, Theorem 7) to convert constraints on the statement instances $\mathbf{x}$ to constraints on the schedule coefficients $\mathbf{c}^{\mathrm{x}}$, $\mathbf{c}^{\mathrm{n}}$ and c^{c}.

Lemma 1 (Affine form of the Farkas Lemma) *Given a system of affine inequality constraints $A\,\mathbf{x} + \mathbf{b} \geq \mathbf{0}$ that admits at least one solution, then any affine inequality $\mathbf{c} \cdot \mathbf{x} + d$ is valid for all solutions of this system iff it can be written as a non-negative affine combination of the inequality constraints in the system, i.e.,*

$$\mathbf{c} \cdot \mathbf{x} + d \equiv \lambda_0 + \boldsymbol{\lambda}^{\mathrm{T}}(A\,\mathbf{x} + \mathbf{b}) \quad \textit{with } \lambda_0, \boldsymbol{\lambda} \geq 0. \tag{10}$$

The elements of $\boldsymbol{\lambda}$ and λ_0 are called the Farkas multipliers.

By equating the constant term and the coefficients of $\mathbf{x}$ on both sides of (10) and taking into account the constraints $\lambda_0, \boldsymbol{\lambda} \geq 0$, a system is obtained that describes all solutions for the schedule coefficients $\mathbf{c}$ and d. The Farkas multipliers $\boldsymbol{\lambda}$ and λ_0 serve as existentially quantified variables in this system. It is customary, but not required, to eliminate these existentially quantified variables using Fourier-Motzkin elimination before looking for solutions to this system. For example, Lim and Lam (1997) eliminate the Farkas multipliers. The `isl` scheduler also performs this elimination.

Note that when the Farkas multipliers are eliminated, then what is known as the Farkas algorithm (Feautrier 1992a, Section 3.2) becomes essentially identical to the vertex method (Feautrier 1992a, Section 3.1). In particular, the variables (i.e., $\mathbf{c}$ and d) and the set of solutions for these variables are the same. The only potential difference is the algorithm for computing the constraints on $\mathbf{c}$ and d. However, a worst-case exponential number of constraints in the result cannot be avoided, because the final number of constraints cannot be smaller than the number of non-redundant generators for the input. Balev et al. (1998) compare this vertex method to the original Farkas algorithm (in terms of the Farkas multipliers).

In `isl` the set of schedule constraints may be described as a Presburger set (Verdoolaege 2016). Any existentially quantified variables or integer divisions in this description are eliminated using Fourier-Motzkin elimination prior to the application of the Farkas lemma. This amounts to an overapproximation of the schedule constraints and therefore also to potentially more restrictive constraints on the schedule coefficients. After elimination, the set of schedule constraints may still be described as a union of sets of the form that can be used as input to the application of the Farkas lemma. In this case, the Farkas lemma is applied to each of these sets separately and the final result is the intersection of the results of the individual applications. This result is equivalent to the result of applying the Farkas lemma to the closed convex hull of the original set of schedule constraints.

6.4 The Feautrier Scheduler in `isl`

Even though the Feautrier scheduler is only used as a fallback scheduler for the Pluto scheduler in `PPCG`, it is discussed first because it is somewhat easier to explain. The Feautrier scheduler implemented in `isl` is essentially the one described by Feautrier (1992b, Section 4). The basic idea is to carry as many dependences as possible in each level of the schedule. In the original context, where the core step is repeated until all dependences have been carried, this leads to a schedule of small depth. When used as a fallback for the Pluto scheduler, it means that as many of the possibly problematic dependences as possible are removed.

The remainder of this section first describes how to obtain constraints on schedule coefficients in general. Then it describes the LP problem that is constructed from these constraints on schedule coefficients and how the solution is

used. Finally, the choice of the groups of schedule constraints that is used in the construction of the LP problem is explained. After solving the LP problem, a new single-dimensional band is added to the schedule tree computed so far and all schedule constraints that are carried by this band are removed from consideration. If the Feautrier scheduler was used as a fallback for the Pluto scheduler, then the Pluto scheduler is run again with the updated set of schedule constraints.

6.4.1 Constraints on schedule coefficients

The first step is to convert all schedule constraints to constraints on the schedule coefficients using the Farkas lemma. These constraints are then combined into a single LP problem and solved using a *lexicographic* LP solver (Feautrier 1988b) implemented in `isl`. The use of a lexicographic solver allows several objective functions to be specified that are considered in order. By default, the scheduler computes a lexicographically *minimal* solution. Note that the solver of Feautrier (1988b) is also a *parametric* LP solver, but this feature is not used during the computation of schedules.

The primary objective function tries to maximizes the number of (groups of) schedule constraints that are carried by the schedule by incurring a cost when the group of schedule constraints is *not* carried. For this purpose, an additional variable e_i is introduced for each group i of schedule constraints, along with the constraints $0 \leq e_i \leq 1$ and

$$e_i \leq f(\mathbf{b}) - f(\mathbf{a}), \tag{11}$$

valid for each schedule constraint $\mathbf{a} \to \mathbf{b}$ in the group, where f is the affine schedule function (9) that is being computed. That is, e_i can only be strictly larger than 0 if *all* schedule constraints in the group are carried by f. A cost of 1 is incurred when the group is not carried, i.e., when $e_i = 0$, and no cost is incurred when e_i can be made equal to 1. That is, the first objective function is

$$\sum_i (1 - e_i)\,. \tag{12}$$

The fact that an additional variable needs to be introduced for each group explains why only groups of schedule constraints can be carried and not individual schedule constraints. Note that despite the fact that the problem is (initially) solved as a rational LP problem, it can be shown that whatever the solution, e_i is either zero or one (and not some rational value in between) (Feautrier 1992b, Lemma 5). Also note that it is this objective function that drives the LP solver to produce a non-trivial solution. That is, without this objective function, the solver would tend to produce a schedule with all zero coefficients.

While it is crucial to obtain a non-trivial schedule, the schedule coefficients should not be too large either. The secondary objective is then to minimize the size of the coefficients. This objective is broken up into two objective functions. The first is the sum of all the coefficients of symbolic constants

$$\sum_{i,j} c^{\mathrm{n}}_{i,j}, \tag{13}$$

with $c^{\mathrm{n}}_{i,j}$ the coefficient of symbolic constant n_j in node i. The second is the sum of all the coefficients of the variables in absolute value. Note that it is sufficient to only consider non-negative values for the coefficients of the symbolic constants since a schedule with a negative such coefficient $c^{\mathrm{n}}_{i,j}$ can be replaced by an equivalent schedule where $-c^{\mathrm{n}}_{i,j}$ is added to the coefficient of n_j in each node of the schedule constraint graph. However, for the actual variables, negative

coefficients should be allowed. There are at least two more or less straightforward ways of allowing negative coefficients while minimizing their absolute values. The first option is to allow the coefficients themselves to attain negative values and to introduce extra variables that are constrained to be greater than or equal to both a specific coefficient and its opposite. The objective function is then the sum of these extra variables. The second option is to encode each coefficient $c^{\mathrm{x}}_{i,j}$ as the difference between two non-negative coefficients $c^{\mathrm{x},+}_{i,j} - c^{\mathrm{x},-}_{i,j}$. The objective function is then

$$\sum_{i,j} \left(c^{\mathrm{x},+}_{i,j} + c^{\mathrm{x},-}_{i,j} \right). \tag{14}$$

Since the solver of Feautrier (1988b) operates on non-negative variables by default, `isl` takes this second option.

6.4.2 LP problem

The order of the variables in the LP problem is then as as described below. Note that the lexicographic LP solver computes the lexicographically minimal value of these variables. That is, each variable is considered as an objective function, in the order in which the variables appear.

1. sum of uncarried groups (12)
2. sum of symbolic constant coefficients (13)
3. sum of variable coefficients (14)
4. for each group i
   - e_i
5. for each node i
   - the coefficients of the variables in reverse order, i.e., for each variable x_j
     - $c^{\mathrm{x},-}_{i,j}$
     - $c^{\mathrm{x},+}_{i,j}$
   - the coefficients of the symbolic constants $c^{\mathrm{n}}_{i,j}$
   - the coefficient of the constant term c^{c}_i

The variables e_i are not placed in any particular order, even though the order can affect which groups will end up getting carried. For the coefficients of the variables, the negative part is placed first in order to prefer positive coefficients. The order in which the pairs of coefficients are placed is the opposite of the order of the corresponding variables. This favors zero coefficients for later variables. Note that since the optimization is mainly driven by the number of carried groups of schedule constraints, the order of the coefficients is usually not very important. The order is mainly chosen for consistency with the order in the ILP problem of Section 6.5.5 below. The coefficient of the constant term c^{c}_i can be assumed to be non-negative for the same reason that the coefficients of the symbolic constants can be assumed to be non-negative. Note that prior to `isl-0.18-688-g927d231e47`, the coefficients of the variables were not stored in reverse order, while prior to `isl-0.18-703-g393c656e7d`, the coefficients of the symbolic constants were stored in front of those of the variables.

Besides the implicit non-negativity constraints, the constraints $e_i \leq 1$ and the equality constraints expressing the sums in the first three objective functions,

all remaining constraints are obtained through applications of the Farkas lemma. In general, a group i of schedule constraints relates instances of two fixed nodes j and k, with the statement instances represented by two sequences of variables, $\mathbf{x}$ and $\mathbf{y}$. The constraints may further reference some symbolic constants $\mathbf{n}$. Applying the Farkas lemma produces constraints on the coefficients of a generic affine function

$$g(\mathbf{c}^{\mathrm{x}}, \mathbf{c}^{\mathrm{y}}, \mathbf{c}^{\mathrm{n}}, d; \mathbf{x}, \mathbf{y}) = \mathbf{c}^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}^{\mathrm{y}} \cdot \mathbf{y} + \mathbf{c}^{\mathrm{n}} \cdot \mathbf{n} + d \tag{15}$$

that is valid for the set of schedule constraints, i.e., with

$$g(\mathbf{c}^{\mathrm{x}}, \mathbf{c}^{\mathrm{y}}, \mathbf{c}^{\mathrm{n}}, d; \mathbf{x}, \mathbf{y}) \geq 0. \tag{16}$$

The constraints on these coefficients then need to be translated into constraints on the schedule coefficients that appear in the LP problem. In particular, the schedule function f in (11) may have different coefficients per node, i.e.,

$$f_k(\mathbf{y}) - f_j(\mathbf{x}) - e_i \geq 0, \tag{17}$$

with

$$\begin{aligned} f_j(\mathbf{x}) &= \mathbf{c}_j^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}_j^{\mathrm{n}} \cdot \mathbf{n} + c_j^{\mathrm{c}} \\ &= (\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}) \cdot \mathbf{x} + \mathbf{c}_j^{\mathrm{n}} \cdot \mathbf{n} + c_j^{\mathrm{c}}. \end{aligned} \tag{18}$$

That is, the schedule coefficients need to satisfy

$$(\mathbf{c}_k^{\mathrm{x},+} - \mathbf{c}_k^{\mathrm{x},-}) \cdot \mathbf{y} - (\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}) \cdot \mathbf{x} + (\mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}}) \cdot \mathbf{n} + c_k^{\mathrm{c}} - c_j^{\mathrm{c}} - e_i \geq 0. \tag{19}$$

This is a special case of the generic constraint (16), i.e.,

$$g(-(\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{x},+} - \mathbf{c}_k^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}}), c_k^{\mathrm{c}} - c_j^{\mathrm{c}} - e_i; \mathbf{x}, \mathbf{y}) \geq 0. \tag{20}$$

The constraints obtained through the application of the Farkas lemma can therefore be added to the LP problem through an appropriate mapping of the coefficient variables. When j is different from k, i.e., when the schedule constraints involve two distinct nodes, then this substitution is a simple matter of copying coefficients, with some sign changes.

For schedule constraints that relate instances of a node j to other instances of the same node, the same mechanism can be used, although the substitution is slightly more involved. In the `isl` implementation, a slightly different mechanism is used for such intra-node schedule constraints that exploits the fact that the two instances of schedule coefficients are the same from the start. In particular, the Farkas lemma is in this case not applied to the set of schedule constraints itself, but to its *difference set*, i.e., the set containing $(\mathbf{b} - \mathbf{a})$ for each schedule constraint $\mathbf{a} \to \mathbf{b}$. The generic valid constraint (16) in terms of these differences is then of the form

$$g(\mathbf{c}^{\delta}, \mathbf{c}^{\mathrm{n}}, d; \boldsymbol{\delta}) \geq 0 \tag{21}$$

with

$$g(\mathbf{c}^{\delta}, \mathbf{c}^{\mathrm{n}}, d; \boldsymbol{\delta}) = \mathbf{c}^{\delta} \cdot \boldsymbol{\delta} + \mathbf{c}^{\mathrm{n}} \cdot \mathbf{n} + d. \tag{22}$$

In the intra-node case, the constraint (11) is of the form

$$f_j(\mathbf{y}) - f_j(\mathbf{x}) - e_i \geq 0, \tag{23}$$

with f_j as in (18). That is, the schedule coefficients need to satisfy

$$(\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}) \cdot (\mathbf{y} - \mathbf{x}) - e_i \geq 0. \tag{24}$$

This is a special case of the generic constraint (21), i.e.,

$$g((\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), \mathbf{0}, -e_i; \boldsymbol{\delta}) \geq 0. \tag{25}$$

```
  for (i = 0; i < _PB_N; i++) {
     //j<i
     for (j = 0; j < i; j++) {
        for (k = 0; k < j; k++) {
A:         A[i][j] -= A[i][k] * A[j][k];
        }
B:      A[i][j] /= A[j][j];
     }
     // i==j case
     for (k = 0; k < i; k++) {
C:      A[i][i] -= A[i][k] * A[i][k];
     }
D:   A[i][i] = SQRT_FUN(A[i][i]);
  }
```

Listing 14: Excerpt from `PolyBench/C 4.1 cholesky` benchmark, with additional statement labels

6.4.3 Solution

While the variables e_i are guaranteed to attain only the integral values 0 or 1 in the solution of the LP problem, the actual coefficients may have non-integral values in this solution. In this case, the numerators with respect to a common denominator are used as the actual schedule coefficients. The common denominator may, however, be very large, resulting in (undesirable) large schedule coefficients. When the Feautrier scheduler is used as a fallback for the Pluto scheduler, the LP problem is therefore turned into an ILP problem if the solution of the LP problem turns out be non-integral. Note that this switch to an ILP problem was introduced in `isl-0.18-679-g6e75a0dd15`.

Example 9 *Consider the code fragment in Listing 14. At the outermost level, no outer coincidence can be found and the Feautrier scheduler is called to carry schedule constraints. As reported by Zinenko (2017), the rational solution to the corresponding LP problem results in large coefficients in the schedule when the command line option* `-DPOLYBENCH_USE_SCALAR_LB` *is used instead of the command line option* `-DPOLYBENCH_USE_C99_PROTO`*. In particular, the affine schedule computed by the Feautrier scheduler (prior to* `isl-0.18-679-g6e75a0dd15`*) is*

$$\begin{aligned}&\mathtt{A}(i,j,k) \to 1998 + 3993j + 1997k; \mathtt{B}(i,j) \to 1997 + 5991j;\\ &\mathtt{C}(i,k) \to 3994i + 1997k; \mathtt{D}(i) \to 5991i.\end{aligned} \tag{26}$$

The minimal integral solution to the LP problem is

$$\mathtt{A}[i,j,k] \to i + j + k; \mathtt{B}[i,j] \to i + 2j; \mathtt{C}[i,k] \to 2i + k; \mathtt{D}[i] \to 3i, \tag{27}$$

which is similar to the solution that is obtained when specifying the command line option `-DPOLYBENCH_USE_C99_PROTO`*. This solution will be discussed further in Example 16 on page 41. Note that the integral solution carries the same number of groups as the rational solution.*

6.4.4 Groups of schedule constraints

The only remaining issue to be discussed is the selection of the groups of schedule constraints. This selection involves both the choice of which schedule constraints are to be considered, and how this set is subdivided into groups. The

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
S:      s += f(i, j);
```

Listing 15: Code with dependences between consecutive statement instances

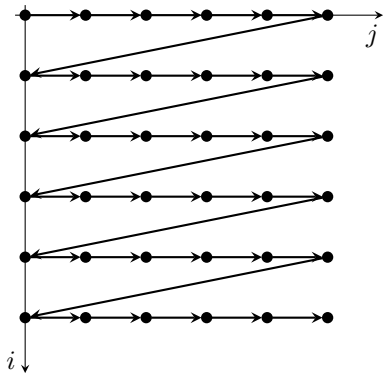

Figure 16: Flow dependences for the code in Listing 15

set of considered constraints clearly needs to include the validity schedule constraints as these constraints need to at least be satisfied for the resulting schedule to be valid. Since there is no mechanism for influencing the conditions of the conditional validity schedule constraints, the corresponding conditioned validity constraints need to be included as well. Finally, if the Feautrier scheduler is used as a backup of the Pluto scheduler with forced outer coincidence, then the coincidence schedule constraints are included too (since `isl-0.18-677-g9a97a1a`).

One way of subdividing the selected schedule constraints into groups would be to take one group for each pair of nodes. However, this means that (11) requires all schedule constraints between a pair of nodes to be carried before the algorithm can make any progress. This turns out to be too restrictive in practice. Instead, the Feautrier scheduler implemented in `isl` takes the disjunctive normal form of the set of schedule constraints between a pair of nodes and introduces a group for each disjunct. Note that when tagged conditional validity schedule constraint are being used, the conditioned validity constraints used by the Feautrier scheduler are those with the tags projected out. The same disjunct may appear several times in this projection and only one group needs to be introduced for each distinct disjunct. (Versions of the scheduler prior to `isl-0.18-675-g757d4ce` would introduce a group for each occurrence.)

Example 10 *As an illustration of the need for subdividing the set of schedule constraints, consider the code shown in Listing 15. The flow dependence relation is*

$$\begin{aligned}\{\, \mathtt{S}[i,j] \to \mathtt{S}[i,j+1] : 0 \le i < n \wedge 0 \le j < n-1; \\ \mathtt{S}[i,n-1] \to \mathtt{S}[i+1,0] : 0 \le i < n-1 \,\},\end{aligned} \tag{28}$$

as shown in Figure 16. The difference set of the flow dependence relation (28) *is*

$$\{\, \mathtt{S}[0,1] : n \ge 2; \mathtt{S}[1,1-n] : n \ge 2 \,\}. \tag{29}$$

Applying the Farkas lemma to each disjunct separately yields

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, c^{\mathrm{n}}, d] : c^{\mathrm{n}} \ge 0 \wedge c_1^{\mathrm{x}} + 2c^{\mathrm{n}} + d \ge 0 \,\} \tag{30}$$

and

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, c^{\mathrm{n}}, d] : c_1^{\mathrm{x}} \leq c^{\mathrm{n}} \wedge c_1^{\mathrm{x}} \leq c_0^{\mathrm{x}} + 2c^{\mathrm{n}} + d \,\}. \tag{31}$$

Note that the (symbolic) constants in (23) *cancel out, which is reflected in the fact that* $\mathbf{c}^{\mathrm{n}}$ *is replaced by* $\mathbf{0}$ *in* (25)*. Performing this substitution results in*

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, d] : c_1^{\mathrm{x}} + d \geq 0 \,\} \tag{32}$$

and

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, d] : c_1^{\mathrm{x}} \leq 0 \wedge c_1^{\mathrm{x}} \leq c_0^{\mathrm{x}} + d \,\}. \tag{33}$$

A group of schedule constraints is only carried when the corresponding e_i *is equal to* 1*, i.e., when* d *is replaced by* -1*, according to* (25)*. Performing this substitution yields*

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}] : c_1^{\mathrm{x}} > 0 \,\} \tag{34}$$

and

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}] : c_1^{\mathrm{x}} \leq 0 \wedge c_1^{\mathrm{x}} < c_0^{\mathrm{x}} \,\}. \tag{35}$$

Clearly, the intersection of these two sets in empty, meaning that it is impossible for all schedule constraints to be carried by a single affine function. In particular, only the second group can be carried. Since $0 \leq e_i \leq 1$ *holds,* d *must satisfy* $-1 \leq d \leq 0$*. This implies that* $c_1^{\mathrm{x}} = 0$ *and* c_0^{x} *can be chosen to be strictly greater than* $c_1^{\mathrm{x}} = 0$*.*

6.5 The Pluto Scheduler in `isl`

The Pluto scheduler constructs a permutable band of independent affine schedule functions that have a small dependence distance over the proximity schedule constraints. Each member of the band is constructed in turn by solving an ILP problem using the same set of schedule constraints, i.e., without removing schedule constraints carried by earlier members. Using the same schedule constraints for all members ensures that these members are permutable within the band. The ILP formulation needs to take into account the various types of schedule constraints. Moreover, it also needs to take care of ensuring independence and, in particular, of avoiding trivial (all-zero) solutions. These issues are discussed next.

6.5.1 Validity schedule constraints

The treatment of validity schedule constraints is similar to their treatment by the Feautrier scheduler in Section 6.4.2, except that no attempt is made to carry any schedule constraints. That is, no variables e_i are introduced and the imposed constraint is simply $f(\mathbf{b}) \geq f(\mathbf{a})$ (3) rather than $f(\mathbf{b}) - f(\mathbf{a}) \geq e_i$ (11). The derivation of the generic constraint (16) that is valid for a group of schedule constraints is the same as in Section 6.4.2, but it is applied to $f_k(\mathbf{b}) \geq f_j(\mathbf{a})$, which results in the special case

$$g(-(\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{x},+} - \mathbf{c}_k^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}}), c_k^{\mathrm{c}} - c_j^{\mathrm{c}}; \mathbf{x}, \mathbf{y}) \geq 0 \tag{36}$$

for inter-node schedule constraints (compare with (20)). As in Section 6.4.1, the coefficients of the variables are written as the differences between two non-negative variables. See Section 6.5.7 below for a further discussion. For intra-node schedule constraints, the special case

$$g((\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), \mathbf{0}, 0; \boldsymbol{\delta}) \geq 0 \tag{37}$$

is obtained (compare with (25)).

Note that unlike the case for the Feautrier scheduler in Section 6.4.4, the groups of validity schedule constraints for which constraints on the schedule coefficients are computed are simply the complete sets of validity schedule constraints between a given pair of nodes. In particular, there is no need to subdivide this set because all schedule constraints need to be respected and there is no specific need to carry any subset of them.

6.5.2 Proximity schedule constraints

Distances over proximity schedule constraints (4) are minimized by imposing a generic uniform bound on those distances and by subsequently minimizing the coefficients of this bound. The uniform bound is of the form

$$u(\mathbf{n}) = \mathbf{m} \cdot \mathbf{n} + m_0 \tag{38}$$

and the constraint imposed on the schedule coefficients is

$$f(\mathbf{b}) - f(\mathbf{a}) \leq \mathbf{m} \cdot \mathbf{n} + m_0, \tag{39}$$

i.e.,

$$-f_k(\mathbf{b}) + f_j(\mathbf{a}) + \mathbf{m} \cdot \mathbf{n} + m_0 \geq 0. \tag{40}$$

The constant m_0 can be assumed to be non-negative since any solution to (39) with a negative value of m_0 remains a solution after replacing this negative value by zero. As to the elements of $\mathbf{m}$, while the symbolic constants $\mathbf{n}$ are *usually* non-negative, this is not guaranteed. The elements of $\mathbf{m}$ may therefore need to take on negative values and are then also written as the differences between two non-negative variables $\mathbf{m}^+ - \mathbf{m}^-$ in the ILP formulation. Identifying (40) with the generic valid constraint (16) and with f_j as in (18), yields the special case

$$g((\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), -(\mathbf{c}_k^{\mathrm{x},+} - \mathbf{c}_k^{\mathrm{x},-}), \mathbf{m}^+ - \mathbf{m}^- - \mathbf{c}_k^{\mathrm{n}} + \mathbf{c}_j^{\mathrm{n}}, m_0 - c_{k,0} + c_{j,0}; \mathbf{x}, \mathbf{y}) \geq 0 \tag{41}$$

for inter-node schedule constraints. For intra-node schedule constraints, the special case of (21) is

$$g(-(\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), \mathbf{m}^+ - \mathbf{m}^-, m_0; \boldsymbol{\delta}) \geq 0. \tag{42}$$

Note that (39) only bounds the schedule constraint distance from above. For any proximity schedule constraint $\mathbf{a} \to \mathbf{b}$, the same schedule coefficient constraints are therefore also added for the pair $\mathbf{b} \to \mathbf{a}$. However, it is fairly common for the proximity schedule constraints to be a subset of the validity schedule constraints. If this is the case then $f(\mathbf{b}) - f(\mathbf{a})$ is already bounded from below, specifically by zero, and then these additional constraints are not introduced.

The uniform bound on schedule constraint distances is the reason why the Pluto scheduler uses an ILP formulation rather than an LP formulation. If the schedule coefficients are not required to be integers, then the schedule coefficients could be arbitrarily scaled down and the uniform bound would no longer be a meaningful measure for proximity. The fact that there is only a single uniform bound also means that if there is any proximity schedule constraint that forces a large value for $u(\mathbf{n})$, i.e., that necessarily has a large distance, then any other proximity schedule constraints are essentially ignored. That is, it does not matter what the corresponding distances are as long as they are smaller than this large distance.

6.5.3 Coincidence schedule constraints

Coincidence schedule constraints $\mathbf{a} \to \mathbf{b}$ indicate a preference of having $f(\mathbf{b}) = f(\mathbf{a})$. They are treated as a pair of validity schedule constraints imposing $f(\mathbf{b}) \geq f(\mathbf{a})$ and $f(\mathbf{a}) \geq f(\mathbf{b})$. However, since coincidence schedule constraints only indicate a preference, a solution where these constraints on the schedule coefficients do not hold is also accepted. In particular, when the construction of a permutable band is initiated, the coincidence schedule constraints are taken into account for the computation of all members, but as soon as this fails to produce a solution, they are disregarded in a new attempt for the computation of the current member and also in the computation of any further members. If the failure occurs for the first member and if forced outer coincidence is requested by the user, then the construction of the permutable band is aborted and the Feautrier scheduler fallback is used instead. Any member that is computed with coincidence schedule constraints in effect is marked as coincident.

6.5.4 Conditional validity schedule constraints

Recall from (4.2) that a collection of conditional validity schedule constraints consists of two parts, the conditions and the conditioned validity schedule constraints. A conditioned validity schedule constraint only needs to be taken into account if there is any adjacent condition that holds, i.e., that has its endpoints scheduled apart. Since there are two ways of satisfying a conditioned validity schedule constraint, either as a regular validity schedule constraint or by ensuring that all adjacent conditions are coscheduled, the `isl` schedule does not initially take them into account in order not to be forced to make a choice. Instead, the conditional validity schedule constraints are checked after the computation of each member of the band.

In particular, after a solution to the ILP problem has been computed, the scheduler checks whether any conditional validity schedule constraint would be violated by this solution. That is, it checks for violated conditioned validity schedule constraints with adjacent conditions that are not coscheduled. If no such combination can be found, then the solution is accepted and the scheduler moves on to the computation of the next member of the band. Otherwise, all such adjacent conditions are forced to be coscheduled by marking them as *local* and the computation of the entire band is restarted. Local conditions are treated in the same way as coincidence schedule constraints are treated in Section 6.5.3, except that they remain forced throughout the entire computation of the band. Note that it is not individual conditions that are marked local, but rather groups of conditions. In particular, if the conditional validity schedule constraints are not tagged, then the entire set of conditions between the same pair of nodes is marked local. If tags are involved, then only those with the same tag spaces are marked local. The computation of the band may be restarted several times as extra conditional validity schedule constraints turn out to be violated, but since there is only a finite number of nodes and tag spaces, this process is guaranteed to terminate.

6.5.5 ILP problem

The ILP problem is solved using the same solver that is used to solve the LP problem of Section 6.4.2, except that it is instructed to compute an integer solution. In particular, a lexicographically minimal solution is computed, meaning that each variable in the ILP problem is considered as an objective function in the order in which the variables appear. Some further handling is required to ensure a non-trivial solution, but this is discussed below in Section 6.5.7.

The order of the variables is as follows.

1. the sum of the coefficients of the symbolic constants in the uniform schedule constraint distance of Section 6.5.2
$$\sum_i m_i^+ + m_i^- \tag{43}$$
2. the constant term of this uniform bound m_0
3. sum of symbolic constant coefficients (13)
4. sum of variable coefficients (14)
5. for each symbolic constant n_i
   - m_i^-
   - m_i^+
6. for each node i
   - the coefficients of the variables in reverse order, i.e., for each variable x_j
     - $c_{i,j}^{\mathrm{x},-}$
     - $c_{i,j}^{\mathrm{x},+}$
   - the coefficients of the symbolic constants $c_{i,j}^{\mathrm{n}}$
   - the coefficient of the constant term $c_{i,0}$

The first objective function favors uniform bounds on schedule constraint distances with small coefficients for the symbolic constants. In the best case, they are all zero and the uniform bound is a constant. The second objective function favors uniform bounds with a small constant term or simply a small uniform bound if it is constant. The third objective function is shared with the LP problem of the Feautrier scheduler in Section 6.4.2. The order in which the pairs of coefficients are placed is the opposite of the order of the corresponding variables. This favors zero coefficients for later variables. Note that prior to `isl-0.18-688-g927d231e47`, the coefficients of the variables were not stored in reverse order, while prior to `isl-0.18-703-g393c656e7d`, the coefficients of the symbolic constants were stored in front of those of the variables.

6.5.6 Linear independence

Each member of the constructed band needs to be made linearly independent of all previous members in the band and of all members in bands nodes higher up in the schedule tree. Without any such treatment, the ILP problems set up for each member would be the same and then also the solutions would be the same. Linear independence is ensured by forcing the solution to be non-zero along directions that are linearly independent of those previous band members. For every node i, let C_i be a matrix with as rows the coefficients $\mathbf{c}_i^{\mathrm{x}}$ of these members. Let
$$H_i = C_i\, U_i \tag{44}$$
be the Hermite normal form (Schrijver 1986, Chapter 4) of H_i, with U_i a unimodular matrix. In particular, all but the first r_i columns of H_i are zero, with r_i the rank of C_i (and of H_i). Taking the transpose of (44) yields
$$H_i^{\mathrm{T}} = U_i^{\mathrm{T}}\, C_i^{\mathrm{T}}, \tag{45}$$

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
  for (i = 1; i<= _PB_N - 2; i++)
    for (j = 1; j <= _PB_N - 2; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                 + A[i][j-1] + A[i][j] + A[i][j+1]
                 + A[i+1][j-1] + A[i+1][j]
                 + A[i+1][j+1])/SCALAR_VAL(9.0);
```

Listing 17: Excerpt from `PolyBench/C 4.1` `seidel-2d` benchmark, with minor reformatting

where all but the first r_i rows of H_i^{T} are zero. The last rows of U_i^{T} are therefore linear combinations of schedule coefficients that are all zero on schedule coefficients that are linearly dependent on the rows of C_i. Conversely, any sequence of schedule coefficients that is linearly independent of the rows of C_i will result in a non-zero value when multiplied with these rows. The mechanism described in Section 6.5.7 below then ensures that the results of multiplying the schedule coefficients with these $d_i - r_i$ rows (with d_i the length of $\mathbf{c}_i^{\mathrm{x}}$) are not all zero. It does so by trying to make each of the linear combinations non-zero in turn. The final rows of U_i^{T} are therefore first normalized such that each row has as many final zeros as possible and such that the first non-zero element in the row is positive. This normalization is performed by elementary row operations, in particular a form of Gaussian elimination that first eliminates the last columns and a change of sign of any row that is not lexicographically positive. Having many final zeros favors schedules that involve early variables, while lexicographically positive rows favor a positive initial coefficient. Finally, the rows are expressed in terms of the differences of non-negative variables $\mathbf{c}_i^{\mathrm{x},+} - \mathbf{c}_i^{\mathrm{x},-}$.

Example 11 *Consider the code fragment shown in Listing 17. For the outer level, no coincidence can be obtained and the Feautrier scheduler produces the affine schedule function*

$$\mathtt{S}[t, i, j] \to 4t + 2i + j. \tag{46}$$

At the next level, C is therefore

$$\begin{bmatrix} 4 & 2 & 1 \end{bmatrix}. \tag{47}$$

The Hermite normal form of this matrix is

$$\begin{bmatrix} 4 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & -2 & -4 \end{bmatrix}. \tag{48}$$

The last $3 - 1 = 2$ rows of the transpose of U are

$$\begin{bmatrix} 0 & 1 & -2 \\ 1 & 0 & -4 \end{bmatrix}. \tag{49}$$

Note that both these rows produce zero when multiplied with (the rows of) C. Performing (reverse) Gaussian elimination yields

$$\begin{bmatrix} -1 & 2 & 0 \\ -1 & 0 & 4 \end{bmatrix}, \tag{50}$$

which is turned into

$$\begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & -4 \end{bmatrix}, \tag{51}$$

by changing the signs of the rows to make them lexicographically positive. Finally, these linear combinations of schedule coefficients are expressed in terms of the non-negative variables $\mathbf{c}_i^{\mathrm{x},+} - \mathbf{c}_i^{\mathrm{x},-}$,

$$\begin{bmatrix} 0 & 0 & 2 & -2 & -1 & 1 \\ 4 & -4 & 0 & 0 & -1 & 1 \end{bmatrix}, \tag{52}$$

recalling that the pairs of non-negative variables are stored in opposite order and that $c_{i,j}^{\mathrm{x},-}$ *appears before* $c_{i,j}^{\mathrm{x},+}$. *After the trivial solution has been found, the first row will (first) be made positive by the mechanism of Section 6.5.7 below. This means either the coefficient of* i *or the coefficient of* t *will be made non-zero. That is, the coefficient of* j *will not be forced to be non-zero by this extra constraint because the first row does not involve this coefficient. A solution that only makes the coefficient of* t *non-zero is both valid and lexicographically smaller than one that also makes the coefficient of* i *non-zero. The second affine schedule function is therefore*

$$\mathtt{S}[t, i, j] \to t. \tag{53}$$

The matrix C *is now*

$$\begin{bmatrix} 4 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{54}$$

with Hermite normal form

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -4 & -2 \end{bmatrix}. \tag{55}$$

The last row of the transpose of U *is now*

$$\begin{bmatrix} 0 & 1 & -2 \end{bmatrix}, \tag{56}$$

which ultimately results in the final affine schedule function

$$\mathtt{S}[t, i, j] \to i. \tag{57}$$

Note that the linear independence constraint is only imposed on those nodes i where $d_i - r_i$ is maximal. The other nodes are necessarily allowed to have some linearly dependent members since, in the end, the total number of members will be the same for all nodes. In particular, if there is still some slack for node i, then the current member is allowed to be linearly dependent on the previous members or even equal to zero. If $d_i = r_i$ for every node, then a complete schedule has been computed and the construction process is terminated.

Note that this way of ensuring linear independence is slightly different from that of Bondhugula, Baskaran, et al. (2008), who compute an orthogonal subspace rather than simply a linearly independent subspace. Also note that no such special treatment is needed for the Feautrier scheduler because a band constructed by the Feautrier scheduler is forced to carry some schedule constraints that have not already been carried by any previous bands.

6.5.7 Non-trivial solutions

In the original formulation of the Pluto scheduler by Bondhugula, Baskaran, et al. (2008), the trivial (all zero) solution is avoided by only considering non-negative coefficients and then enforcing the sum of these coefficients to be strictly greater than zero. Negative coefficients can however be useful sometimes, for

example for reversing the execution order. The original formulation has therefore been extended to allow for negative coefficients by Vasilache et al. (2012), Verdoolaege, Juega, et al. (2013) and Acharya and Bondhugula (2015). Acharya and Bondhugula (2015) compare their approach to that of Vasilache et al. (2012) and show that they use fewer decision variables than Vasilache et al. (2012). A detailed description of the approach mentioned by Verdoolaege, Juega, et al. (2013) was only available in the `isl` source code. The present section synthesizes this description.

The reason given by Bondhugula, Baskaran, et al. (2008) to only consider non-negative coefficients is that in the unrestricted sign case, cutting out zero results in a non-convex search space, which would lead to a combinatorial explosion when the number of nodes is large. The `isl` scheduler *does* in fact explore this non-convex search space, but not exhaustively (once a solution has been found). Although the number of cases considered per node is relatively high, the case-split only kicks in when an actual trivial solution has been found. Due to the presence of validity schedule constraints and proximity schedule constraints, a non-trivial solution for one node will usually force a non-trivial solution for related nodes. This means that the case-splitting typically only kicks in for one (or a few) nodes. In fact, if there are groups of nodes that are largely independent of each other, then it is best to schedule them separately, as explained below in Section 7.3.

In order to understand how the case-splitting is implemented, it should be noted that the non-parametric version of the lexicographic LP solver (Feautrier 1988b) used by the `isl` scheduler is built on top of an *incremental* LP solver (Detlefs et al. 2005). This means that after an initial solution has been computed, additional constraints can be added and removed again.

The general goal is to compute a solution for the ILP problem of Section 6.5.5 that is non-trivial on a sequence of regions of variables. Each of these regions corresponds to the $2\,d_i$ variables that encode the coefficients for the variables for a node i with an associated $(d_i - r_i) \times (2\,d_i)$-matrix of linear combinations of these variables (as computed in Section 6.5.6), at least one of which should be non-zero on the variables. Recall that nodes i where $d_i - r_i$ is smaller than the maximal value are allowed to have a linearly dependent band member and therefore do not have an associated region. Besides this sequence of regions, the solver is also given the number of initial objective functions that need to be optimized before the solution is considered to be "optimal". That is, the solver will not consider additional cases once this number of initial objective functions has been fully optimized. These objective functions are called the *significant* objective functions in the remainder of this section. In particular, the number of significant objective functions is set to two, meaning that reducing the sum of the coefficients of the symbolic constants in the uniform schedule constraint distance of Section 6.5.2 and the constant term of this uniform bound are the main objectives.

The solver then operates as follows. First a solution to the ILP problem is computed without any further constraints. This typically results in an all-trivial solution because the variables in each region are all zero and therefore any linear combination of these variables is also zero. The solver then picks the first trivial region and considers the following $2(d_i - r_i)$ cases

- $r_0(\mathbf{x}) \geq 1$
- $r_0(\mathbf{x}) \leq -1$
- $r_0(\mathbf{x}) = 0 \wedge r_1(\mathbf{x}) \geq 1$
- $r_0(\mathbf{x}) = 0 \wedge r_1(\mathbf{x}) \leq -1$

- $r_0(\mathbf{x}) = 0 \wedge r_1(\mathbf{x}) = 0 \wedge r_2(\mathbf{x}) \geq 1$
- $r_0(\mathbf{x}) = 0 \wedge r_1(\mathbf{x}) = 0 \wedge r_2(\mathbf{x}) \leq -1$
- $\ldots,$

where the $\mathbf{x}$ are the variables in the region and the r_j are the rows in the linear independence matrix. Within each case, the corresponding constraints are added to the ILP problem and a new solution is computed. If this new solution still has trivial regions, then a further case-split is applied. When the ILP problem turns out to have no solution or when a solution has been found, the solver backtracks to the latest choice-point and considers the next case. When all cases have been exhausted, the solver backtracks to the previous choice-point. As soon as a solution has been found, the ILP problem is further constrained to look for *better* solutions. One option for looking for better solutions would be to impose that the first significant objective function with a non-zero value in the best solution so far has a smaller value in any further solutions. The choice made in the current implementation is to force the first significant objective function with a non-zero value in the best solution so far to have a zero value in any further solutions.

Note that there are alternatives for the choice of breaking up the search space. In particular, it would be possible to split up the space into only $d_i - r_i + 1$ cases, e.g., by considering positive values for each row in turn (with non-positive values for previous rows) and an additional case with all non-positive values and a constraint that imposes that the sum is negative. The subdivision used by the current implementation is inspired by the fact that is usually preferred to have non-zero coefficients for the first variable(s). The initial rows of the linear independence matrix can then be made to have non-zero values for as few of the final variable coefficients as possible.

Example 12 *As an example of this mechanism in action, consider the computation of a schedule for*

$$\{\, \mathtt{S}(i,j) : 0 \leq i \leq 10 \,\} \tag{58}$$

with proximity schedule constraints

$$\{\, \mathtt{S}[i,j] \rightarrow \mathtt{S}(1+i,j) : 0 \leq i \leq 9 \wedge 0 \leq j \leq 10 \,\}. \tag{59}$$

The difference set of these proximity schedule constraints is

$$\{\, \mathtt{S}(i = 1, j = 0) \,\} \tag{60}$$

and the set of generic valid constraint coefficients for this set is

$$\{\, [a_0, a_1, d] : a_0 + d \geq 0 \,\}. \tag{61}$$

Specializing this set for the uniform bound (39)*, which in the absence of symbolic constants is simply $f(\mathbf{b}) - f(\mathbf{a}) \leq m$, yields*

$$\{\, [c_0, c_1, m] : -c_0 + m \geq 0 \,\} \tag{62}$$

according to (42)*. The ILP problem as described in Section 6.5.5 is therefore*

$$\{\, [0, m, 0, c_0^+ + c_0^- + c_1^+ + c_1^-, c^{\mathrm{c}}, c_0^-, c_0^+, c_1^-, c_1^+] : -(c_0^+ - c_0^-) + m \geq 0 \,\}. \tag{63}$$

Note that the sums of coefficients of symbolic constants are zero because there are no symbolic constants in this example. Solving the ILP problem initially results in the trivial solution

$$\{\, [0,0,0,0,0,0,0,0,0] \,\}. \tag{64}$$

In this example, there is only one region that needs to be non-trivial and it is the region involving the coefficients c_0 and c_1. Since no schedule rows have been computed so far, the linear combinations that ensure linear independence are simply c_0 and c_1, or, in terms of the non-negative variables encoding these coefficients,

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} c_0^- \\ c_0^+ \\ c_1^- \\ c_1^+ \end{bmatrix}. \tag{65}$$

The solver first considers the case $c_0 \geq 1$, i.e., $c_0^+ - c_0^- > 0$. Adding this constraint, the ILP solution is updated to

$$\{\,[0,1,0,1,0,0,1,0,0]\,\}, \tag{66}$$

i.e., $c_0 = 1$, $c_1 = 0$ and $m = 1$. Next, the solver considers the case $c_0 \leq -1$, but since a solution has been found before, the constraint $m = 0$ is imposed as well to ensure a significantly better solution. The updated ILP problem does not have any solutions and so the solver moves on to the case $c_0 = 0 \wedge c_1 \geq 1$, still with the extra $m = 0$ constraint. In this case, the solution

$$\{\,[0,0,0,1,0,0,0,0,1]\,\} \tag{67}$$

is found, i.e., $c_0 = 0$, $c_1 = 1$ and $m = 0$. This solution is determined to be optimal with respect to the significant objective functions and the search terminates.

The mechanism for avoiding trivial solutions described in this section has essentially been used by the `isl` scheduler from its initial implementation, first described by Verdoolaege, Juega, et al. (2013). The only difference with the current version is that the initial implementation would perform a change of basis with the directions of linear independence each available as a separate variable in the changed basis. While several other aspects of the scheduler have been modified since its initial implementation to resolve some scalability issues (as described in Section 7 below), this mechanism for avoiding trivial solutions does not appear to have caused any such scalability issues and has been left unchanged. The main difference with the approach of Acharya and Bondhugula (2015) is that the removal of trivial solutions is performed outside of the ILP problem, rather than being encoded inside this ILP problem. This means in particular that no additional decision variables need to be added. Note that the encoding of the ILP problem itself does use more variables because the coefficients are encoded as differences of non-negative variables, but the reason for this encoding is the objective function on the sum of the sizes of the coefficients as explained in Section 6.4.1. The trivial solutions avoidance mechanism itself does not require such an encoding. Neither the original Pluto scheduler (Bondhugula, Baskaran, et al. 2008) nor the Pluto+ scheduler (Acharya and Bondhugula 2015) appear to have such an explicit constraint on the sum of the sizes of the coefficients.

The approach of Acharya and Bondhugula (2015) also requires predetermined fixed bounds on the schedule coefficients. While the `isl` scheduler does in some cases impose such bounds, it only does so for the specific reasons explained below in Section 7.5. In general, there does not appear to be an inherent reason for placing a specific bound on the coefficients, even though smaller coefficients are preferred in general.

6.5.8 Termination

This section summarizes the elements that influence the termination of the construction of a permutable band and provides further details. There are two

main reasons for termination, either the schedule is complete or the current ILP problem does not admit a solution. The schedule is complete if it is injective for all nodes that are active in the current branch of the schedule tree under construction. Essentially, this means that in the notation of Section 6.5.6, $r_i = d_i$ for every node i. If the current ILP problem has no solution and if coincidence schedule constraints are in effect, then, as explained in Section 6.5.3, these constraints are first taken out of consideration. The only exception is when no members have been computed so far and forced outer coincidence is required. In this case, the construction of a permutable band simple terminates.

After a (possibly empty) permutable band has been computed, the scheduler takes one of several possible next steps. If the band completes the schedule (in this branch), then it is attached to the tree, all carried schedule constraints are removed and the statements are topologically sorted according to the remaining (conditional) validity schedule constraints. Otherwise, if the band is not empty and if the `--schedule-maximize-band-depth` option is set, then the scheduler checks whether it has found a constraint between nodes in distinct strongly connected components of the schedule constraint graph that may explain why it was impossible to find another member in the band. If there is such a constraint, then the graph is split into two parts, separating the two relevant strongly connected components. The current band is discarded and new bands are computed in each of the two parts. This mechanism is not described in further detail because the detection of such splitting constraints is only based on a heuristic and because the mechanism described in Section 7.3 below is much better at guaranteeing a maximal band depth. Otherwise, if the band is not empty, then the band is attached to the schedule tree, all carried schedule constraints are removed and the next (nested) permutable band is computed. If the band is empty, on the other hand, it is discarded and the Feautrier scheduler fallback is used to carry (and remove) some of the schedule constraints.

6.6 Known Issues

In practice, the combination of the Pluto scheduler with the Feautrier scheduler as fallback seems to work out reasonably well. There are, however, some known issues with these scheduling algorithms, some of which are highlighted here.

6.6.1 Long scheduling times

When there are a large number of nodes, the Feautrier scheduler needs to solve a large LP problem. The Pluto scheduler even needs to solve a large ILP problem. Solving such problems can take a long time, even if just for manipulating the large data structures. One possible solution is to use an off-the-shelf "industrial" ILP solver as proposed by Vasilache et al. (2012), who use the Gurobi ILP solver, and by Acharya and Bondhugula (2015), who use GLPK (Makhorin n.d.). Such solvers are more heavily optimized than those available in `pip` or `isl` and are capable of solving large problems more quickly. In particular, they may perform most if not all computations using floating point operations, while `pip` and `isl` use exact integer arithmetic. They do not typically support the computation of a *lexicographic* optimum, though. A lexicographic optimization criterion therefore needs to be encoded in a single linear objective function, e.g., by multiplication with decreasing powers of a "large" integer value. The large computation time can also be mitigated by reducing the sizes of the (I)LP problems, e.g., by grouping closely related statements in a single statement (see Section 7.4) or, in case of the Pluto scheduler, by performing the scheduling incrementally (see Section 7.3).

6.6.2 Loop coalescing

Both the Feautrier scheduler and the Pluto scheduler can produce schedules that correspond to loop coalescing (Polychronopoulos 1987), where two or more nested loops are combined into a single loop. Loop coalescing can be a useful optimization, but it is undesirable in the output of especially the Pluto scheduler because it distorts the perception of the dimensionality of the schedule. In particular, as explained in Section 6.5.6 and Section 6.5.8, the Pluto scheduler will continue producing band members until the total number of band members is greater than or equal to the dimension of each node under consideration. A band member that performs loop coalescing is then only considered to cover one node dimension, but in reality it covers two or more node dimensions. The Pluto scheduler will then continue adding members (possibly in a separate nested band), which will then appear to be coincident because they only deal with a single instance. Furthermore, these loop coalescing schedules have large coefficients, conflicting with the simplicity optimization criterion. Since the Feautrier scheduler is used as a fallback for the Pluto scheduler in the `isl` scheduler, loop coalescing in the Feautrier scheduler is equally undesirable. The cause of loop coalescing depends on the scheduler. For the Pluto scheduler, the driving force is the set of proximity schedule constraints as the coalescing schedule looks like it brings instances closer to each other. For the Feautrier scheduler, the coalescing schedule is typically capable of carrying more groups of validity schedule constraints.

Example 13 *Consider once more the code in Listing 15 on page 28 from Example 10 on page 28, but assume now that* $\mathtt{n}$ *has a known value, say* 10000*. The scheduler may then produce a band member of the form*

$$\{\, \mathtt{S}[i, j] \to 10000\, i + j \,\}, \tag{68}$$

which carries all schedule constraints. Since the node `S` *is two-dimensional, the scheduler considers the schedule computed so far to be incomplete. Since the coalescing band member* (68) *is not coincident, a second nested band is then constructed with a single member that can be anything that is linearly independent with respect to the linear expression in* (68)*, say*

$$\{\, \mathtt{S}[i, j] \to i \,\}. \tag{69}$$

Since all schedule constraints have been removed, this member is marked coincident, while the corresponding "loop" only iterates over a single instance.

One way of avoiding such coalescing is to impose a bound on the size of the coefficients. This is what both the Pluto tool (Bondhugula, Hartono, et al. 2008) and `Polly` (Grosser, Größlinger, et al. 2012) do. In particular, the latter uses the `isl` scheduler and sets the maximal coefficient to 20 by default using the `--schedule-max-coefficient` option. Even though large coefficients are not desirable in general, imposing such an arbitrary bound does not sound like a very elegant solution. Furthermore, it fails on directions with small sizes and it cannot be applied to the Feautrier scheduler (when computing rational solutions). Section 7.5 describes a more tailored approach, including a mechanism for dealing with coalescing in the Feautrier scheduler.

6.6.3 Infeasibility caused by proximity schedule constraints

While proximity schedule constraints are only intended to assign an order of preference to the valid schedules, the presence of an "excessive" number of proximity schedule constraints can result in the ILP problem used by the Pluto

```
A:   a = f1();
     for (int i = 0; i < n; ++i)
B:       A[i] = a;
C:   b = f1();
     for (int i = 0; i < m; ++i)
D:       B[i] = b;
```

Listing 18: Example with too many proximity schedule constraints

scheduler to become infeasible due to the way they are handled in Section 6.5.2. In particular, it may be impossible to find a uniform bound on the proximity schedule constraint distances (39) if there are simply too many such proximity schedule constraints.

Example 14 *Consider the code fragment in Listing 18 and assume that no general bounds on* `m` *and* `n` *are available. The proximity schedule constraints are*

$$\{ \mathtt{A}[] \to \mathtt{B}[i] : 0 \leq i < n; \mathtt{C}[] \to \mathtt{D}[i] : 0 \leq i < m \}. \tag{70}$$

An affine schedule f that is non-trivial for `B` *will schedule* $\mathtt{B}[0]$ *and* $\mathtt{B}[n-1]$ *at a distance of at least $n-1$. Since they both need to be scheduled after* $\mathtt{A}[]$*, at least one of $f(\mathtt{B}[0]) - f(\mathtt{A}[])$ or $f(\mathtt{B}[n-1]) - f(\mathtt{A}[])$ is necessarily greater than or equal to $n-1$. In particular, the uniform bound on the schedule constraint distances $u(m,n)$ needs to be greater than or equal to $n-1$. Due to the schedule constraints between* `C` *and* `D`*, the uniform bound $u(m,n)$ similarly needs to be greater than or equal to $m-1$. On the other hand, the constraints $\{ \mathtt{A}[] \to \mathtt{B}[i] : 0 \leq i < n$ are effective for* every *value of* `m`*, since the description of these constraints does not contain any reference to* `m`*. This means that*

$$f(\mathtt{B}[i]) - f(\mathtt{A}[]) \leq u(m,n) \tag{71}$$

needs to hold for every value of `m`*, including both arbitrarily large positive values and arbitrarily large negative value. This in turn means that the coefficient of m in the affine expression $u(m,n)$ needs to be zero and similarly for the coefficient of n due to the constraints between* `C` *and* `D`*. The expression $u(m,n)$ cannot be guaranteed to be larger than m and n without involving m or n. In other words, the scheduler will not find any non-trivial solutions.*

Example 14 represents an extreme case, but similar, if slightly more obscure, cases where this phenomenon takes place do occur in practice. The `isl` scheduler does not currently have any specific mechanism for dealing with such issues. The Pluto scheduler will simply fail to find a solution and the Feautrier scheduler fallback will be used instead. Since the Feautrier scheduler does not take proximity schedule constraints into account, it will not suffer from this problem, but overall the `isl` scheduler may miss tilable bands and/or coincident members. It may be possible to remove specific patterns of groups of proximity schedule constraints that are overly constraining, but a general solution may not be that easy to achieve because the inability to impose a uniform bound may be due to sequences of proximity schedule constraints.

For the specific case of Example 14, the incremental mode of the scheduler described in Section 7.3 provides a means to circumvent the problem as the two independent pieces of the schedule constraint graph will be scheduled independently. Also, if both `m` and `n` are known to be non-negative in the code of Example 14, then a uniform upper bound can easily be determined. Even if these variables can attain negative values, it is customary for at least some

bounds on `m` and `n` to be known, if only those derived from the types of these variables in the source code. It used to be recommended to not include such bounds in the schedule constraints because it could lead to the coalescing issue described in Section 6.6.2, but due to the mechanism described in Section 7.5 below, this should no longer in itself be a reason to exclude such bounds.

6.6.4 The Feautrier scheduler carries too much

While the use of the Feautrier scheduler as a fallback for the Pluto scheduler appears to work out reasonably well in practice, it does constitute a slight abuse of this Feautrier scheduler. In particular, while the Feautrier scheduler was designed to carry as many schedule constraints as possible, it tends to carry a bit too many constraints for the purpose for which it is used in the `isl` scheduler.

Example 15 *An illustration of this effect was already shown in Example 8 on page 21. In particular, when using the standard Feautrier scheduler to carry schedule constraints between* $\mathtt{S}[t,i,j]$ *and* $\mathtt{T}[t,i,j]$*, it produces the schedule function* (8)*, reproduced here,*

$$\mathtt{S}[t,i,j] \to 2t; \mathtt{T}[t,i,j] \to 2t+1. \tag{72}$$

That is, as instructed, it carries schedule constraints between instances of $\mathtt{S}[t,i,j]$ *and* $\mathtt{T}[t,i,j]$ *with the same value of* t*, as well as between instances of* $\mathtt{T}[t,i,j]$ *and those of* $\mathtt{S}[t,i,j]$ *with the next value of* t*. However, in practice it is sufficient to only carry one of these two groups, which can be accomplished without introducing the coefficient* 2*.*

Section 7.1 describes a simple technique for detecting the commonly occurring pattern of Example 15. However, it cannot recover from more complicated cases of overzealous dependence constraint carrying by the Feautrier scheduler.

Example 16 *Consider once more the code fragment in Listing 14 on page 27. As explained in Example 9 on page 27, no outer coincidence can be found at the outermost level and the Feautrier scheduler is called to carry schedule constraints. The affine schedule computed by the Feautrier scheduler is*

$$\mathtt{A}[i,j,k] \to i+j+k; \mathtt{B}[i,j] \to i+2j; \mathtt{C}[i,k] \to 2i+k; \mathtt{D}[i] \to 3i. \tag{73}$$

Note that this affine function does not have the simple pattern $m \cdot f_j(\mathbf{i}) + c_j$ *with a fixed* m *for all nodes* j*. The simple technique of Section 7.1 is therefore unable to simplify this affine schedule. Essentially, the common factor* 3 *is spread out over* $i+j+k$*,* $i+2j$ *or* $2i+k$ *with* $k < j < i$*, in three of the statements.*

Section 7.6 below describes a simple technique for preventing the needless carrying that occurs in Example 16.

6.7 Forced Outer Coincidence Alternatives

While the combination of the Pluto scheduler with the Feautrier scheduler as fallback seems to work out reasonably well in practice, Section 6.6.4 illustrates that this combination is not ideal in all cases. It is therefore useful to look at some potential alternatives.

6.7.1 Select part of original schedule

By design, the Feautrier scheduler fallback typically gets called when no outer parallelism is available based on the currently active schedule constraints. In such cases, the desired schedule function that should be used to carry some of the active schedule constraints is usually fairly obvious, at least at a conceptual level, and often also corresponds to a part of the original schedule. One possible alternative would therefore be to start off from the original schedule and to first try and construct a new top-level band taking into account all schedule constraints, but if this fails, then to move down in the original schedule tree and then to try again with the schedule constraints not carried by the top-level band(s). Once a new band has been constructed, it would be tricky to try and extract information from the original schedule tree to recover from a nested band without outer coincidence, but since `PPCG` only exploits the outermost band with coincidence, there may be no need to try and enforce coincidence on nested bands. At this point, this potential approach has not been evaluated yet. However, it will clearly fail to expose parallelism that is not already available in the input code, as in Example 17 below.

6.7.2 Single non-coincident member bands

An alternative that has seen some preliminary testing, is to not terminate the construction of a band when it cannot have any coincident members, but instead to allow a single (non-coincident) member to be constructed and to terminate the construction of the band after this single member has been created. A nested band is then constructed with the schedule constraints carried by the single-member band removed. The advantage of this approach is that it does not suffer from the overeager schedule constraint carrying of the Feautrier scheduler described in Section 6.6.4. However, this same difference in behavior also results in a failure to expose parallelism at inner levels on some inputs. This means that there is nothing for `PPCG` to map to an accelerator. The approach with the Feautrier scheduler fallback is therefore still preferred for now, even though there is no guarantee that mapping part of the code to an accelerator necessarily results in faster code.

Example 17 *Continuing from Example 11 on page 33, recall that the Feautrier scheduler produces the affine schedule function*

$$\mathtt{S}[t, i, j] \to 4t + 2i + j. \tag{74}$$

This affine function has coefficients greater than one, but it is able to carry so many schedule constraints that the nested band can have two coincident members. The alternative described in this section produces

$$\mathtt{S}[t, i, j] \to t \tag{75}$$

instead for the outermost node. This affine schedule does not carry enough schedule constraints to allow coincident members in the next band. Instead, the same mechanism triggers twice more resulting in the nested single non-coincident member bands

$$\mathtt{S}[t, i, j] \to i \tag{76}$$

and

$$\mathtt{S}[t, i, j] \to j. \tag{77}$$

That is, the original schedule is reproduced.

Note that when this alternative is combined with the incremental scheduling of Section 7.3 below, then care needs to be taken not to merge a band constructed in this way with any other band.

6.7.3 Validity ILP solver

Another alternative that has seen some preliminary testing, is to use the same Pluto scheduler ILP problem from Section 6.5.5 to create a single-member band, but to ignore all schedule constraints except the validity schedule constraints. As in the case of the previous alternative, the main driver is not to carry as many validity schedule constraints as possible, but to compute a non-trivial affine schedule. This affine schedule will necessarily carry at least some of the validity schedule constraints. Overall the behavior of this alternative is similar to that of the previous alternative, being unable to extract parallelism on some inputs where the Feautrier scheduler succeeds. The set of benchmarks where this occurs is slightly different, though.

Example 18 *For the `PolyBench/C 4.1 cholesky` benchmark, also discussed in Example 16 on page 41, the alternative approach of Section 6.7.2 produces the outer band*

$$\mathtt{A}[i,j,k] \to k; \mathtt{B}[i,j] \to j; \mathtt{C}[i,k] \to k; \mathtt{D}[i] \to i, \tag{78}$$

while the approach of this section produced the outer band

$$\mathtt{A}[i,j,k] \to i; \mathtt{B}[i,j] \to i; \mathtt{C}[i,k] \to i; \mathtt{D}[i] \to i, \tag{79}$$

The first results in some inner bands with coincident members, while the second does not. See also Example 34 on page 66. On the other hand, for the `PolyBench/C 4.1 ludcmp` benchmark, the approach of this section exposes some (if very limited) parallelism, while the approach of Section 6.7.2 does not.

7 Improvements

This section describes some improvements on top of the core scheduler described in Section 6. Note that this core scheduler has also evolved since it was introduced in `isl-0.06-43-g1192654`, but those changes have been integrated in the description in Section 6. For example, schedule trees were only introduced in `isl-0.14-249-gb0c84f7`, but there is little point in describing the original schedule representation in this report. The changes in this section are those that can easily be described separately.

7.1 Unscaling

As explained in Section 6.6.4, the Feautrier scheduler fallback has a tendency to carry more schedule constraints than strictly needed for its (slightly improper) use by the `isl` scheduler, in particular unnecessarily introducing coefficients greater than one. This section describes a simple technique for detecting and handling a commonly occurring pattern, which was first introduced in `isl-0.06-162-gf4d7d23`. In most cases, this technique has been superseded by the technique of Section 7.6 below, but it can still be useful when this latter technique is not effective.

If there are n statements that depend on each other in sequence, while the first statement in turn depends on the previous iteration of the last statement, then the Feautrier scheduler tends to produce a schedule with a factor n and different offsets for the individual statements in order to carry the validity schedule constraints between each pair of successive statements. If the `--schedule-split-scaled` option is set (enabled by default), then the `isl` scheduler therefore looks for this specific pattern in the output of the Feautrier scheduler. In particular, it looks for a common divisor m among all the schedule

coefficients (other than the constant terms) that is greater than one and divides the schedule by this common divisor, rounding down the constant terms to the nearest integers. That is, if the original affine functions are of the form

$$m \cdot f_i(\mathbf{x}) + c_i, \tag{80}$$

then they are replaced by

$$f_i(\mathbf{x}) + \lfloor c_i/m \rfloor . \tag{81}$$

The remainders of the constant terms, i.e., $c_i \bmod m$ can be used to construct a sequence node with children according to increasing values of these remainders. The relative order imposed by this combination of scaled down band node and sequence node is the same as that imposed by the original band node, with the benefit that the factor m has been removed from the band node. The original implementation of this feature would effectively construct such a sequence node, but since `isl-0.18-704-g68cd880`, this sequence node is no longer constructed. The scaled down band node already ensures that it is *possible* to order the statements in this way and there is no need to *enforce* this particular order. If needed, the same or a similar order can be obtained as the strongly connected components of the remaining statement-level schedule constraint graph, but some of these components can also be merged if this turns out to be profitable.

Example 19 *Recall from Example 8 on page 21 that the outer band schedule computed by the Feautrier scheduler for the code fragment shown in Listing 11 on page 21 is*

$$\mathtt{S}[t,i,j] \to 2t; \mathtt{T}[t,i,j] \to 2t+1. \tag{82}$$

The common divisor of the schedule coefficients (ignoring constant terms) is 2*. This band is therefore replaced by the band*

$$\mathtt{S}[t,i,j] \to t; \mathtt{T}[t,i,j] \to t. \tag{83}$$

The scheduler is then free to combine instances of $\mathtt{S}[t,i,j]$ *and* $\mathtt{T}[t,i,j]$ *that share a common* t*-value based on its optimization criteria.*

7.2 Domain Compression

The instance sets of some statements may satisfy some equality constraints, causing the actual dimensions of the instance sets to be different from those of their ambient spaces. As described in Section 6.5.6, the total number of members in bands along a path from root to leaf in the schedule tree needs to be greater than or equal to the number of corresponding schedule coefficients, which is equal to the dimension of the ambient space. If the actual dimension of the instance set is smaller, then an excessive number of band members may be computed, with the extra inner members appearing to be coincident while actually only scheduling a single instance. This mismatch in dimension is similar to what happens in the loop coalescing schedules described in Section 6.6.2. The solution implemented in `isl` is to *compress* the ambient spaces such that their dimensions match the actual dimensions of the instance sets. This domain compression was first introduced in `isl-0.13-167-g8241654`.

Domain compression is performed by first computing, for each statement, the integer affine hull of its instance set in order to determine the equality constraints satisfied by this instance set. Variable compression (Meister 2004) is then applied to these equality constraints in order to obtain a bijective mapping from a lower-dimensional space to the affine hull in the original space. In particular, let the equality constraints be given as

$$-C(\mathbf{n}) + M\mathbf{x} = 0, \tag{84}$$

with $\mathbf{n}$ the symbolic constants. Let H be the Hermite normal form (Schrijver 1986, Chapter 4) of M, i.e.,

$$\begin{bmatrix} H_1 & 0 \end{bmatrix} = H = M\,U = M\,\begin{bmatrix} U_1 & U_2 \end{bmatrix} \qquad \text{or} \qquad M = H\,Q = H\,\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, \quad (85)$$

with H_1 lower triangular, $Q = U^{-1}$ a unimodular matrix, U split column-wise into U_1 and U_2 in the same way H is split into H_1 and 0, and Q accordingly split row-wise into Q_1 and Q_2. Let $\mathbf{x}'_1 = Q_1\mathbf{x}$ and $\mathbf{x}'_2 = Q_2\mathbf{x}$, then $\mathbf{x}'_1$ is fixed by the equality constraints (84), i.e.,

$$\mathbf{x}'_1 = H_1^{-1}C(\mathbf{n}), \quad (86)$$

and $\mathbf{x}$ can be written as

$$\mathbf{x} = U_1H_1^{-1}C(\mathbf{n}) + U_2\mathbf{x}'_2, \quad (87)$$

with $\mathbf{x}'_2$ representing the lower-dimensional space. The inverse transformation is simply

$$\mathbf{x}'_2 = Q_2\mathbf{x}. \quad (88)$$

Finally, the `isl` scheduler computes schedule coefficients for these compressed variables $\mathbf{x}'_2$ rather than for the original variables $\mathbf{x}$.

If domain compression is applied, then the equality constraints (84) of the corresponding affine hulls are added to all corresponding schedule constraints. This is needed in case there are schedule constraints that relate instances outside the instance set, specifically outside the affine hull that was used in the compression. Since the scheduler only computes coefficients for the compressed variables, it can never carry any schedule constraints that relate elements outside these affine hulls. An alternative would be to simply intersect both sides of the schedule constraints with the instances set, but when this feature was introduced, this could have caused a lot of additional loop coalescing. Now that the countermeasure for loop coalescing from Section 7.5 below are available, this decision could be reconsidered.

Example 20 *Consider the code fragment in Listing 19 on the next page. The outermost* `t`*-loop performs the exact same computation several times.* `PPCG`*'s dead code elimination detects that each iteration overwrites the data written by the previous iteration and removes all but the last iteration of this loop. By default, the statement instance sets constructed by* `pet` *have a coordinate for each outer loop in the input code. After dead code elimination, the first coordinates in the statement instance sets therefore have a fixed value. Without domain compression, the scheduler could schedule this coordinate as a coincident band member since the single instance is completely independent of any other instance. This would cause* `PPCG` *to make the wrong decisions about which parts should be mapped to the accelerator. The domain compression successfully removes the fixed coordinate from the statement instance sets, thereby solving this problem. Note that this particular benchmark has been removed from later versions of* `PolyBench/C`*.*

7.3 Incremental Scheduling

7.3.1 Motivation

The original implementation of the Pluto scheduler in `isl` with respect to fusion is based on the original description of Bondhugula, Baskaran, et al. (2008, Section 3.7). In particular, an entire weakly connected component is treated

```
for (t = 0; t < _PB_NITER; t++) {
  for (j = 0; j <= _PB_MAXGRID - 1; j++)
    for (i = j; i <= _PB_MAXGRID - 1; i++)
      for (cnt = 0; cnt <= _PB_LENGTH - 1; cnt++)
        diff[j][i][cnt] = sum_tang[j][i];

  for (j = 0; j <= _PB_MAXGRID - 1; j++) {
    for (i = j; i <= _PB_MAXGRID - 1; i++) {
      sum_diff[j][i][0] = diff[j][i][0];
      for (cnt = 1; cnt <= _PB_LENGTH - 1; cnt++)
        sum_diff[j][i][cnt] =
          sum_diff[j][i][cnt - 1] + diff[j][i][cnt];
      mean[j][i] = sum_diff[j][i][_PB_LENGTH - 1];
    }
  }

  for (i = 0; i <= _PB_MAXGRID - 1; i++)
    path[0][i] = mean[0][i];

  for (j = 1; j <= _PB_MAXGRID - 1; j++)
    for (i = j; i <= _PB_MAXGRID - 1; i++)
      path[j][i] = path[j - 1][i - 1] + mean[j][i];
}
```

Listing 19: Excerpt from `PolyBench/C 3.2` `reg_detect` benchmark, with minor reformatting

as a whole and it is only split into (groups of) strongly connected components when the combined problem does not admit a solution. The handling is somewhat similar to the "maxfuse" option (Bondhugula 2008, Section 4.1.2), as the `isl` scheduler cuts the statement-level schedule constraint graph into two parts. However, it only does so when the `--schedule-maximize-band-depth` option is set, in which case it discards the entire band and tries again after splitting the graph. The choice of where to cut is also somewhat different, which the `isl` scheduler tries to determine (probably not very successfully) from the failed attempt to find a solution. When the `--schedule-maximize-band-depth` is not set, the `isl` scheduler simply moves on to the next band when the current band can no longer be extended. The original Pluto scheduler also has a "smartfuse" option (Bondhugula 2008, Section 4.1.2), which is not supported by the `isl` scheduler. This option also starts out from the entire component, but uses a different mechanism for deciding how to break up the schedule constraint graph. Yet another mechanism is used by the "wisefuse" option of Mehta, Lin, et al. (2014), but it also starts out from the entire component.

The splitting performed in case of `--schedule-maximize-band-depth` does not actually guarantee that the bands will have more members after splitting. Furthermore, there was no mechanism for trying to split the statement-level schedule constraint graph in order to (attempt to) increase the number of coincident members in the bands. The incremental scheduling, first introduced in `isl-0.16.1-11-ge3e8120`, tries to remedy this situation by first computing a band for each strongly connected component *separately* and then combining the results incrementally. That is, instead of trying to guess how to break up the schedule constraint graph in order to increase the number of (coincident) band members, the strongly connected components are only combined if these

numbers do not decrease. This is similar to how Lim, Liao, et al. (2001, Section 5) greedily combine component schedules if they share some reuse and if the combination does not eliminate parallelism. The incremental scheduling is now the default and can be *disabled* using the `--schedule-whole-component` option.

As an additional advantage, when the incremental scheduling kicks in, the number of statements in a strongly connected component is smaller than the number of statements in the entire graph. This means that the corresponding ILP problems have fewer variables. Even though multiple such ILP problems need to be solved by the incremental scheduler, with additional ILP problems to be solved for combining (groups of) strongly connected components, the reduced number of variables usually results in a net win.

The incremental scheduler solves a long-standing issue with the core scheduler of Section 6 in that it would occasionally lose parallelism by fusing too much. The `--schedule-serialize-sccs` option can prevent such undesirable fusion but only because it prevents *any* fusion. The issue was first reported by Juega (2011) on the `PolyBench/C 2mm` benchmark and was also briefly discussed by Verdoolaege, Juega, et al. (2013). For this `2mm` benchmark, the undesired fusion can coincidentally also be prevented through the `--schedule-maximize-band-depth` option (enabled by default), because the fused band has only two members, while a maximal band has three members. However, as shown in Example 21 on page 49 below, there are other cases where the number of coincident members needs to be targeted specifically.

7.3.2 Overview

The incremental scheduler only kicks in when there is more than one strongly connected component in the statement-level schedule constraint graph. The first step consists of computing a band using the Pluto scheduler of Section 6.5 for each such strongly connected component. Note that the Feautrier scheduler fallback is not yet used at this point, meaning that some of the computed bands may have zero members.

The scheduler then tries to incrementally combine *clusters* of strongly connected components, where each cluster initially consists of a single strongly connected component. Three choices are involved in this incremental combination

- which clusters to combine,
- how to combine them,
- which combinations to accept.

7.3.3 Clusters to combine

As to the choice of which clusters to combine, first note that a cluster with an empty band is never combined with any other cluster, unless the schedule is already complete for all statements involved. That is, a cluster with an empty band is only eligible if the reason for the band being empty is not that the constructed ILP problem did not have a solution, but rather that there was no need to construct an ILP problem in the first place.

Furthermore, clusters are only combined with each other if there is at least one proximity schedule constraint between the two (distinct) clusters. In particular, if the user did not specify any proximity schedule constraint that connects the two clusters, then there is no need to bring instances in one cluster close to instances in the other cluster and there is therefore no need to combine the

two clusters. If there are proximity schedule constraints between more than a single pair of statements in distinct clusters, then the order in which they are considered is based on the weight of each such group of proximity schedule constraints between a pair of statements. The weight is an estimate of the number of coordinates that could still be completely aligned when looking only at the group of proximity schedule constraints. In particular, it is computed as the number of equality constraints between input and output coordinates in the relation describing the group of proximity schedule constraints, after projecting out directions that are already handled by outer nodes in the schedule tree. The reasoning behind this heuristic is that the schedule constraint distance could in theory be minimized to zero over this many band members along these equality constraints. If there is more than one group with the same maximal weight, then the distance between the clusters involved in a topological sort of the original clusters is used to determine which pair of clusters to combine, where the smallest distance is preferred. Adjacent or near-adjacent clusters are preferred because intermediate clusters may be forced to be combined in as well when two non-adjacent clusters are being combined.

7.3.4 Combining clusters

While the pair of clusters to be combined is chosen based on proximity schedule constraints, there may also be indirect validity schedule constraints between the two clusters. In particular, this may happen when the two clusters are not adjacent in the topological order. Any intermediate cluster needs to be considered together with the two chosen clusters to ensure validity of the schedule. These intermediate clusters are obtained by computing the strongly connected component of the statement-level schedule constraint graph with extra virtual edges between statements in the two chosen clusters to ensure that they end up in the same strongly connected component. All clusters that are determined to belong to this strongly connected component are combined together. If there is any group of proximity schedule constraints between any two statements in distinct clusters among this selected set of clusters that has already been considered before, then the combination attempt is aborted.

A new ILP problem is then constructed that schedules the selected clusters with respect to each other. In this ILP problem, the "dimension" of each cluster is taken to be the number of members in the current band node associated to the cluster. Scheduling clusters with respect to each other means that any choice taken in the construction of these clusters is preserved. If there are any proximity schedule constraints inside any of the clusters with a large distance, then the scheduler will not make any attempt to make the schedule constraint distances across clusters any smaller than this large distance because proximity schedule constraints are handled using a single uniform bound as described in Section 6.5.2. It may therefore make sense to perform an additional translation step to bring clusters closer to each other, but this is not currently performed by the `isl` scheduler.

An alternative to scheduling clusters with respect to each other would be to schedule all the individual statements in the clusters with respect to each other. This would allow for more scheduling freedom as choices taken in earlier schedules computed for parts of the clusters are no longer fixed, but are instead completely ignored. The downside of such an approach is that the number of variable in the ILP problem would be proportional to the total number of statements rather than to the total number of clusters involved.

```
  for (i = 0; i < _PB_M; i++)
     for (j = 0; j < _PB_N; j++) {
        for (k = i+1; k < _PB_M; k++)
A:         B[i][j] += A[k][i] * B[k][j];
B:      B[i][j] = alpha * B[i][j];
     }
```

Listing 20: Excerpt from `PolyBench/C 4.1 trmm` benchmark, with additional statement labels

7.3.5 Acceptable combinations

Once a band for combining clusters has been computed, what is left to determine is whether to accept the result. This choice is described below. If the result is accepted, then the band that was computed relative to the clusters is applied to the bands of the clusters themselves and the clusters are combined into a single cluster. If the combination is rejected, then the group of proximity schedule constraints that elicited this attempt is marked such that it will no longer be selected in any subsequent merge attempts.

The choice of whether to accept or reject a cluster schedule is based on the following criteria. If the cluster band is empty, i.e., if it has zero members, then it is rejected. If the `--schedule-maximize-band-depth` option is set and the band is not complete, meaning that the number of members in the combined result would be lower than the number of members in at least one of the current clusters, then the cluster schedule is rejected. If the `--schedule-maximize-coincidence` option is set and the number of coincident members in the schedule band is smaller than the maximal number of coincident members in the cluster bands, then the cluster schedule is rejected. Finally, at least one group of proximity schedule constraints needs to be optimized "completely" before the combination is accepted.

A group of proximity schedule constraints is considered to have been optimized completely if the resulting schedule constraint distances are bounded by a small number in all coordinate directions. In particular, this small number is currently taken to be 2. If the dimension of the band is greater than the number of coordinate directions that can be expected to be optimized by the edge (as determined by its weight), then the distances are allowed to be unbounded in the remaining coordinate directions, but only if either the source or the destination has a fixed value in that direction. This allows a statement that produces values that are used by several instances of another statement to be merged with that other statement. However, merging such clusters will introduce an inherently large proximity distance inside the merged cluster, meaning that proximity distances will no longer be optimized in subsequent merges. These merges are therefore only allowed after all other possible merges have been examined.

Example 21 *Consider the code fragment show in Listing 20. If the user sets the `--schedule-whole-component` option, then the schedule shown in Listing 21 on the following page is computed. Note that the single band node has only one coincident member. Without this option, the schedule shown in Listing 22 on the next page is computed instead. In particular, the nodes `A` and `B` each form a strongly connected component and a schedule band is first computed for each component separately. Since there are proximity schedule constraints connecting the two nodes, an attempt is made to combine the two components. The resulting schedule is similar to the one shown in Listing 21 on the following page. In particular, the combined band has only one coincident member,*

```
domain: "[n, m] -> { A[i, j, k] : i >= 0 and 0 <= j < n and i < k < m;
        B[i, j] : 0 <= i < m and 0 <= j < n }"
child:
  schedule: "[n, m] -> [{ B[i, j] -> [(j)]; A[i, j, k] -> [(j)] },
        { B[i, j] -> [(m)]; A[i, j, k] -> [(k)] },
        { B[i, j] -> [(i)]; A[i, j, k] -> [(i)] }]"
  permutable: 1
  coincident: [ 1, 0, 0 ]
  child:
    sequence:
    - filter: "[n, m] -> { B[i, j] }"
    - filter: "[n, m] -> { A[i, j, k] }"
```

Listing 21: Whole-component schedule for the code in Listing 20 on page 49

```
domain: "[n, m] -> { A[i, j, k] : i >= 0 and 0 <= j < n and i < k < m;
        B[i, j] : 0 <= i < m and 0 <= j < n }"
child:
  sequence:
  - filter: "[n, m] -> { A[i, j, k] }"
    child:
      schedule: "[n, m] -> [{ A[i, j, k] -> [(j)] },
        { A[i, j, k] -> [(k)] }, { A[i, j, k] -> [(i)] }]"
      permutable: 1
      coincident: [ 1, 0, 0 ]
  - filter: "[n, m] -> { B[i, j] }"
    child:
      schedule: "[n, m] -> [{ B[i, j] -> [(i)] },
        { B[i, j] -> [(j)] }]"
      permutable: 1
      coincident: [ 1, 1 ]
```

Listing 22: Incremental schedule for the code in Listing 20 on page 49

while the band for `B` *separately has two coincident members. Assuming that the* `--schedule-maximize-coincidence` *option is set, this combination is therefore rejected and the two components are scheduled with respect to each other using a sequence node.*

7.3.6 Termination

The process of combining clusters continues until there is only one cluster left or until all groups of proximity schedule constraints have been considered (possibly for a second time). If there is more than one cluster left, then a sequence node is introduced that schedules the remaining clusters in their topological order. Within each child of this sequence node, the schedule is extended as explained in Section 6.5.8. In particular, strongly connected components with an empty band will not have been merged with any other clusters and will be handled by the Feautrier scheduler at this point.

7.4 Grouping

As mentioned in Section 6.6.1, solving the (I)LP problems can take a long time if many statements are involved. This section describes a technique for reducing the number of statements in the input of the scheduler in some commonly occurring cases. This means that the technique does not modify the scheduler itself, but rather the *user* of the scheduler, i.e., `PPCG`. In particular, sequences of statements in the original schedule are grouped together if the schedule con-

straints indicate that they depend on each other. That is, the grouping technique takes into account both the original schedule and the schedule constraints. This grouping was first introduced in `ppcg-0.05-103-gc7d7a17`. It is enabled by default and can be disabled using the `--no-group-chains` command line option.

Grouping of statements that are executed in sequence in the original program is not uncommon in polyhedral compilation. For example, `Graphite` (Trifunovic et al. 2010) in GCC and `Polly` (Grosser, Größlinger, et al. 2012) in LLVM both use basic blocks as indivisible units. Mehta and Yew (2015) propose a very similar grouping into "O-molecules", which consist of successive statements with the same enclosing loops. The disadvantage of such purely syntactic forms of grouping is that they prevent any form of loop distribution on statements that appear within the same basic block in the input program. The grouping described in this section, by contrast, takes into account both syntactic and dependence information and only groups statements together that are likely to end up being grouped together by the scheduler anyway. That is, while it does necessarily remove some scheduling freedom, it tries to do so only when this extra scheduling freedom is unlikely to be exploited.

The main motivation for the grouping described in this section is a commonly occurring pattern where a sequence of statements compute some intermediate results that are used in the next statement. Typically, these intermediate results are stored in scalar variables, but the storage locations are not a factor for deciding whether to group or not. Instead, the decision is based only on syntactical information and on dependence information. In terms of syntactical information, two statements to be merged need to be adjacent leaf node children in a sequence node in the schedule tree representing the original execution order. The dependence information used during grouping is the intersection of the collection of validity schedule constraints with the collection of proximity schedule constraints between the two statements. This intersection is a binary relation between the instance spaces of the two statements and needs to satisfy the following conditions:

- the relation needs to cover both statements completely,
- it needs to be a bijection, and
- it needs to relate instances that are coscheduled by the original schedule.

That is, two statements are only grouped together if they are executed in sequence and if the first statement produces values that are used by exactly the instance of the second statement that follows immediately after, and this for all instances of both statements.

The intersection of validity schedule constraints and proximity schedule constraints is used because the resulting pairs are those that should preferably be executed close to each other while the second has to be executed after the first. The relation is required to cover both statements to ensure that the dependence does not just occur for one or a few instances. This test is implemented by checking that the instance sets of the two statements are a subset of the domain and the range of the relation, respectively. The relation is required to be bijective to rule out cases where a value produced by the first statement is used by many instances of the second statement. This test is not strictly needed because a relation failing this test would also fail the next test. In particular, if an instance of the first statement is related to many instances of the second statement then they cannot all be coscheduled with that instance of the first statement. The coscheduling test is needed to ensure that it is not just the statements that appear together, but it is also the specific instances that are executed after each

```
k = (SCALAR_VAL(1.0)-EXP_FUN(-alpha))*
     (SCALAR_VAL(1.0)-EXP_FUN(-alpha))/
     (SCALAR_VAL(1.0)+SCALAR_VAL(2.0)*alpha*
                      EXP_FUN(-alpha)-
               EXP_FUN(SCALAR_VAL(2.0)*alpha));
a1 = a5 = k;
a2 = a6 = k*EXP_FUN(-alpha)*(alpha-SCALAR_VAL(1.0));
a3 = a7 = k*EXP_FUN(-alpha)*(alpha+SCALAR_VAL(1.0));
a4 = a8 = -k*EXP_FUN(SCALAR_VAL(-2.0)*alpha);
b1 =  POW_FUN(SCALAR_VAL(2.0),-alpha);
b2 = -EXP_FUN(SCALAR_VAL(-2.0)*alpha);
```

Listing 23: Excerpt from `PolyBench/C 4.1 deriche` benchmark, with minor reformatting

other that depend on each other. Grouping such adjacent instances is always valid, while grouping non-adjacent instances may not be valid in general. The test is performed using the *prefix schedules* at the two leaves. This prefix schedule is essentially the concatenation of all outer band nodes. Any two instances in the relation need to be mapped to the same values by these prefix schedules. Since the band nodes of the original schedule correspond to the loops in the input program, this means that they need to be executed in the same iterations of the outer loops. Note that this test explicitly refers to the schedule and does not simply compare the instance sets because the instance set is only meant to identify the instances and does not (explicitly) carry any information about the execution order.

When the grouping of two statements is successful, the group is compared with the adjacent (groups of) statements. Two groups of statements are combined if there is at least one pair of statements, one from each group, that satisfies the above criteria. In the implementation, the consecutive leaf node children in a sequence node are compared from last to first. If two leaves are combined into a single leave, then it is compared again with the next leave. For example, if there are three leaves A, B and C, then the pair (B, C) may not satisfy the criteria, but if the pair (A, B) does get merged into a single group, then it may still be possible to combine this group with C, in particular if the pair (A, C) satisfies the criteria.

After the groups of statements have been identified, a mapping from statement instances to group instances is constructed. For the statements involved in any grouping, this mapping is constructed from the prefix schedules, with the identifier of the range set to a unique identifier for the corresponding group. For statements not involved in any grouping, the mapping is simply the identity mapping on their statement instance sets. The combined mapping is applied to the two sides of all schedule constraints (taking into account the tags when appropriate). These modified schedule constraints are then used as input to the `isl` scheduler. In particular, a schedule is computed for the groups of statements. The resulting schedule is then expanded by attaching an expansion node to all leaves that expands the group instances to the original statement instances. Within this expansion node, sequence nodes are attached that order the statements within each group.

Example 22 *Consider the code fragment shown in Listing 23 on page 52, which is only a small part of the actual benchmark. The first statement writes to the scalar* `k`*, which is read by the second statement. Since both statements only have a single instance, the conditions for grouping apply. After this grouping, the*

```
for (j = 0; j < _PB_M; j++)
  {
    mean[j] = SCALAR_VAL(0.0);
    for (i = 0; i < _PB_N; i++)
      mean[j] += data[i][j];
    mean[j] /= float_n;
  }

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_M; j++)
    data[i][j] -= mean[j];

for (i = 0; i < _PB_M; i++)
  for (j = i; j < _PB_M; j++)
    {
      cov[i][j] = SCALAR_VAL(0.0);
      for (k = 0; k < _PB_N; k++)
        cov[i][j] += data[k][i] * data[k][j];
      cov[i][j] /= (float_n - SCALAR_VAL(1.0));
      cov[j][i] = cov[i][j];
    }
```

Listing 24: Excerpt from `PolyBench/C 4.1` `covariance` benchmark

third statement, which also reads from `k`*, becomes adjacent to a group containing the statement writing to* `k` *and is included in the same group. The same applies to the fourth and fifth statement, in that order.*

Example 23 *The code fragment in Listing 24 on the next page illustrates that the intermediate data does not need to be stored in a scalar variable. In particular, the last statement in the last loop nest reads data from the element* `cov[i][j]`*, which is written by the previous statement in the same iteration of the outer loops. The two statements are therefore grouped into a single statement. The generated schedule after grouping is virtually identical to a schedule obtained without grouping. However, the last statement could in theory be split off into a separate loop nest, so the grouping does remove some scheduling freedom.*

Example 24 *An at first sight somewhat surprising case where statement grouping applies is on the code fragment shown in Listing 25 on the following page. Even though statements* `A` *and* `B` *are each guarded by an extra condition, it turns out that these conditions are redundant with respect to the outer loop bounds. This means that there are schedule constraints between* `A` *and* `B` *for each instance of* `A` *and for each instance of* `B`*, i.e., without any exceptions. These two statements are then combined into a group called* `G_0` *and the schedule constraints are reformulated in terms of this group. Note that the instances of this group have different values from those of the original two statements because, by default,* `pet` *uses the values of the outer loop iterators to identify the statement instances, while the group instances correspond to the execution order of these loops. These are different in this case because the outermost loop is executed from high values of* `i` *to low values of* `i`*.*

The schedule that is generated using the modified schedule constraints is shown in Listing 26 on page 55. Note that the outermost band node was created

```
 for (i = _PB_N-1; i >= 0; i--) {
  for (j=i+1; j<_PB_N; j++) {

   if (j-1>=0)
A:    table[i][j] = max_score(table[i][j],
                              table[i][j-1]);
   if (i+1<_PB_N)
B:    table[i][j] = max_score(table[i][j],
                              table[i+1][j]);

   if (j-1>=0 && i+1<_PB_N) {
     /* don't allow adjacent elements to bond */
     if (i<j-1)
C:      table[i][j] = max_score(table[i][j],
                table[i+1][j-1]+match(seq[i], seq[j]));
     else
D:      table[i][j] = max_score(table[i][j],
                              table[i+1][j-1]);
   }

   for (k=i+1; k<j; k++) {
E:    table[i][j] = max_score(table[i][j],
                table[i][k] + table[k+1][j]);
   }
  }
 }
```

Listing 25: Excerpt from `PolyBench/C 4.1 nussinov` benchmark, with additional statement labels and minor reformatting

```
domain: "[n] -> { C[i, j] : i >= 0 and 2 + i <= j < n;
        E[i, j, k] : i >= 0 and j < n and i < k < j;
        D[i, j = 1 + i] : 0 <= i <= -2 + n;
        G_0[i0, i1] : i0 <= 0 and -i0 < i1 < n }"
child:
  schedule: "[n] -> [{ D[i, j] -> [(1)]; C[i, j] -> [(-i + j)];
        E[i, j, k] -> [(-i + j)]; G_0[i0, i1] -> [(i0 + i1)] }]"
  child:
    schedule: "[n] -> [{ D[i, j] -> [(j)]; C[i, j] -> [(j)];
                E[i, j, k] -> [(j)]; G_0[i0, i1] -> [(i1)] },
        { D[i, j] -> [(0)]; C[i, j] -> [(0)]; E[i, j, k] -> [(k)];
                G_0[i0, i1] -> [(0)] }]"
    permutable: 1
    coincident: [ 1, 0 ]
    child:
      sequence:
      - filter: "[n] -> { G_0[i0, i1] }"
      - filter: "[n] -> { D[i, j] }"
      - filter: "[n] -> { E[i, j, k] }"
      - filter: "[n] -> { C[i, j] }"
```

Listing 26: Schedule for the code in Listing 25 on page 54 after grouping

using the Feautrier scheduler and that the unscaling of Section 7.1 happens to be effective in this case. The schedule that is created based on the original schedule constraints (without grouping) has a slightly different form, where the technique of Section 7.1 is not effective, but this appears to be just a coincidence. After expansion of the schedule in Listing 26, the schedule in Listing 27 on page 56 is obtained. This schedule does not only contain an expansion of the group `G_0`*, but also an "expansion" of the statement* `D`*. This is an artifact of the way the schedule is expanded. In particular, an identity expansion is applied to all statements not involved in grouping. These identity expansions are then simplified away if they are trivial, but this simplification process fails to detect the expansion of* `D` *as a trivial expansion due to the equality constraint among the instance identifiers.*

Besides the benchmarks shown in the previous examples, grouping also kicks in on the `PolyBench/C` `correlation` benchmark.

Example 25 *Listing 28 on page 57 shows a schematic overview of a fork of OPAL, an OpenCL Lattice Boltzmann Method solver by Obrecht et al. (2015). The fork was modified to be able to serve as input to* `PPCG` *and was sent to the authors by Ho (2017) under the BSD license. It is available as* `src_ppcg/main.c` *in the attached* `opal.tar.gz`*. The original code has a sequential outer loop with three nested parallel loops. This also turns out to be the best loop structure for this application, so in principle there is no need to compute a new schedule. On the other hand,* `PPCG` *should be able to compute a schedule for such applications, even if just to confirm that no better schedule can be obtained. It is also useful to be able to optimize applications that have similar features but where the original loop structure is not optimal. Without grouping, the* `isl` *scheduler is unable to compute a new schedule for this application. The Pluto scheduler still runs fine, if slowly, but since it cannot find outer coincidence, the* `isl` *scheduler resorts to the Feautrier scheduler, which cannot complete because the LP problem requires too much memory.*

The grouping mechanism manages to group all the dozens of statements at the end of the loop body (not shown in Listing 28 on the next page) because they are all connected through temporary computations. It also includes the statements guarded by the non-static condition, because the entire `if`*-statement*

```
domain: "[n] -> { C[i, j] : i >= 0 and 2 + i <= j < n;
        E[i, j, k] : i >= 0 and j < n and i < k < j;
        D[i, j = 1 + i] : 0 <= i <= -2 + n;
        G_0[i0, i1] : i0 <= 0 and -i0 < i1 < n }"
child:
  schedule: "[n] -> [{ D[i, j] -> [(1)]; C[i, j] -> [(-i + j)];
        E[i, j, k] -> [(-i + j)]; G_0[i0, i1] -> [(i0 + i1)] }]"
  child:
    schedule: "[n] -> [{ D[i, j] -> [(j)]; C[i, j] -> [(j)];
                E[i, j, k] -> [(j)]; G_0[i0, i1] -> [(i1)] },
        { D[i, j] -> [(0)]; C[i, j] -> [(0)]; E[i, j, k] -> [(k)];
                G_0[i0, i1] -> [(0)] }]"
    permutable: 1
    coincident: [ 1, 0 ]
    child:
      sequence:
      - filter: "[n] -> { G_0[i0, i1] }"
        child:
          contraction: "[n] -> { B[i, j] -> G_0[(-i), (j)];
                A[i, j] -> G_0[(-i), (j)] }"
          expansion: "[n] -> { G_0[i0, i1] -> B[i = -i0, j = i1];
                G_0[i0, i1] -> A[i = -i0, j = i1] }"
          child:
            sequence:
            - filter: "[n] -> { A[i, j] }"
            - filter: "[n] -> { B[i, j] }"
      - filter: "[n] -> { D[i, j] }"
        child:
          contraction: "[n] -> { D[i, j] -> D[(i), (1 + i)] }"
          expansion: "[n] -> { D[i, j] -> D[i' = i, j' = 1 + i] }"
      - filter: "[n] -> { E[i, j, k] }"
      - filter: "[n] -> { C[i, j] }"
```

Listing 27: Expanded schedule for the code in Listing 25 on the previous page

```
for (unsigned step = 0; step < MAX_SMALL_STEPS; step++) {
    for (unsigned z = 0; z < 128; z++) {
        for (unsigned y = 0; y < 128; y++) {
            for (unsigned x = 0; x < 128; x++) {
                /* local variable declarations */
                /* ... */
                int g = 0;
                #define f(i) f##i

                if (x == 128-1) g = g | G_W1;
                if (x == 0)     g = g | G_W2;
                if (y == 128-1) g = g | G_W3;
                if (y == 0)     g = g | G_W4;
                if (z == 128-1) g = g | G_W5;
                if (z == 0)     g = g | G_W6;

                f(0) = Lattices[step%2][z][y][x][0];
                if (g & G_BND) {
                    /* ... */
                } else {
                    /* ... */
                }

                /* dozens of statements */
            }
        }
    }
}
```

Listing 28: Schematic overview of a fork of OPAL, an OpenCL Lattice Boltzmann Method solver

```
for (int l = 0; l < 50; l += 1) {
  for (int m = 0; m < 64; m += 1) {
    for (int n = 0; n < 8; n += 1) {
      for (int p = 0; p < 8; p += 1) {
        for (int i = 0; i < 16; i += 1) {
          for (int j = 0; j < 5; j += 1) {
            for (int k = 0; k < 5; k += 1) {
              A[l][(i * 1)][(j * 1 + n)][(k * 1 + p)] +=
                B[l][m][n][p] * C[m][i][j][k];
            }
          }
        }
      }
    }
  }
}
```

Listing 29: Example from Akilesh (2017), with minor reformatting

is seen as a single statement encapsulating dynamic control, and the statement immediately preceding this `if`*-statement. For better or for worse, the only statements that are not included in the group are the statements shown in Listing 28 on page 57 that have a lower-dimensional instance set because they are only executed for a specific value of* `x`*,* `y` *or* `z`*. The reason these statements do not get included is that they do not meet the criterion that the corresponding schedule constraints need to cover the entire instance sets of the other statements involved. After grouping, the* `isl` *scheduler manages to compute a schedule without running out of memory. Example 36 on page 67 shows the effect of various changes to the scheduler on the scheduler execution time for this input.*

7.5 Loop Coalescing Avoidance

As explained in Section 6.6.2, band members that represent loop coalescing are undesirable because they have large coefficients and because they result in a skewed view of the dimensionality of the schedule. One way of avoiding such loop coalescing band members is to impose a bound on the schedule coefficients. The `isl` scheduler allows users to impose such a bound using the `--schedule-max-coefficient` option, but this option is not used by `PPCG` by default because it is not clear which bound should be imposed. While large coefficients are undesirable in general, they may in some cases still be acceptable if this results in coincident band members in the next band as in Example 11 on page 33. On the other hand, picking an upper bound that is too large will not help avoiding loop coalescing when the instance sets have small sizes. Furthermore, such a bound can only be imposed on the Pluto scheduler and not on the Feautrier scheduler and it seems unnecessary to impose such a bound if there is no risk for loop coalescing. This section describes a more tailored approach that was first introduced in `isl-0.16.1-230-gf8f45df`. It may be turned on or off using the `--schedule-treat-coalescing` option and is enabled by default.

Example 26 *Consider the code fragment shown in Listing 29, which was generated by the DSL compiler Latte (Truong et al. 2016). Since the code was automatically generated, it may be possible to change the generation process to produce loops with parametric bounds, but it is still interesting to look at the example as it is. Note in particular that the bounds on the innermost loops are*

fairly small such that a generic bound on the schedule coefficients may not be able to prevent loop coalescing.

7.5.1 Characterization

A band member represents loop coalescing if it reduces the dimension of the statement instance sets, i.e., if it combines two or more instance set coordinate directions into a single dimension. That is, the values of those instance set coordinates can be uniquely extracted from the function value of the band member. Note that this is different from mere skewing, where two or more coordinate directions are also combined, but several combinations of values of the coordinate directions involved map to the same function value such that it is impossible to uniquely extract the original values from the function value.

Example 27 *Recall the band member* (68)

$$\{ \mathtt{S}[i,j] \to 10000\,i + j \} \tag{89}$$

from Example 13 on page 39. Since $0 \leq j < 10000$*, the value of* j *can be extracted as*

$$(10000\,i + j) \bmod 10000, \tag{90}$$

while the value of i *can be extracted as*

$$\lfloor (10000\,i + j)/10000 \rfloor . \tag{91}$$

If the affine schedule function of the band member had been $i + j$*, i.e., a mere skewing, then it would be impossible to uniquely extract* i *or* j *from the function value.*

If there are no holes in the instance set, then one of the coordinates needs to be skewed by the size of the domain in the other coordinate directions to be able to extract both values through division with remainders. That is, coalescing is introduced if

$$\left| \frac{c_i}{c_j} \right| \geq S_j, \tag{92}$$

with c_i the schedule coefficient for coordinate i and S_j the size of the instance set in coordinate direction j. Note that this can only happen if there actually is some fixed upper bound S_j on the size. In particular, if the size is a symbolic constant and if there is no fixed bound on this symbolic constant, then this form of coalescing cannot occur. The precise meaning of the size S_j is described in Section 7.5.2 below. Also note that condition (92) only describes the condition for *introducing* coalescing. If for some reason the instance set is already skewed by that much in the input, then a trivial affine schedule function will also represent coalescing, but this will not be detected by any of the measures described below.

While the condition (92) reflects true coalescing, near coalescing, where the skewing factor is close to S_j, is equally bad. Prohibiting the condition (92) may in effect force the scheduler to look for such near coalescing solutions, where the skewing factor is one less than the size. If the schedule constraints do not connect instances at the extremes of the domain, then a solution with a skewing factor that is even slightly smaller may also be found. To avoid these near coalescing situations, the condition (92) is relaxed to

$$\left| \frac{c_i}{c_j} \right| > \left\lceil \frac{S_j}{2} \right\rceil . \tag{93}$$

The risk of introducing loop coalescing is the reason why PPCG only uses the context very sparingly. If this context assigns a fixed value to the symbolic constants or even imposes a fixed upper bound on them, then intersecting the schedule constraints with the context will copy this fixed value or fixed upper bound to the schedule constraints and increase the risk of loop coalescing. Even just simplifying the schedule constraints with respect to the context may have the same effect. Now that countermeasures for loop coalescing have been put in place, the use of the context can be reconsidered.

Example 28 *Given schedule constraints*

$$\{ [i,j] \to [i+1,j] : 0 \leq i, i+1, j < n \} \tag{94}$$

and a context

$$\{ : n = 1024 \}, \tag{95}$$

intersecting the schedule constraints with the context yields

$$\{ [i,j] \to [i+1,j] : 0 \leq i, i+1, j < 1024 \wedge n = 1024 \}, \tag{96}$$

while simplifying the schedule constraints with respect to the context (computing the "gist") yields

$$\{ [i,j] \to [i+1,j] : 0 \leq i, i+1, j < 1024 \}. \tag{97}$$

In both cases, a fixed upper bound in introduced which could be abused by the scheduler to produce a loop coalescing band member.

7.5.2 Size computation

The loop coalescing condition (92) and the near loop coalescing condition (93) refer to the size of the instance set in a given coordinate direction. This size is taken to be the difference in values for that coordinate for fixed values of the other coordinates. That is, if I is the instance set, then the size S_i in coordinate direction i is

$$\max \{ \delta : \exists \mathbf{x}, x_i' : \mathbf{x} \in I \wedge (x_0, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots x_{d-1}) \in I \wedge \delta = x_i' - x_i \}. \tag{98}$$

If this set of differences does not have a maximum, then S_i is set to ∞. Note that if any domain compression has been applied as in Section 7.2, then the size of the *compressed* instance set is computed.

Example 29 *Consider the instance set*

$$\{ [x,y] : x \geq 0 \wedge x \leq y \wedge x + y \leq 100 \} \tag{99}$$

show in Figure 30 on page 61. The size in the x-direction is

$$\begin{aligned} \max \{ \delta : \exists x, y, x' : & x \geq 0 \wedge x \leq y \wedge x + y \leq 100 \wedge \\ & x' \geq 0 \wedge x' \leq y \wedge x' + y \leq 100 \wedge \delta = x' - x \}, \end{aligned} \tag{100}$$

which is equal to 100*. The maximum is attained for* $x = 0$*,* $y = 0$ *and* $x' = 100$*. The size in the y-direction is*

$$\begin{aligned} \max \{ \delta : \exists x, y, y' : & x \geq 0 \wedge x \leq y \wedge x + y \leq 100 \wedge \\ & x \leq y' \wedge x + y' \leq 100 \wedge \delta = y' - y \}, \end{aligned} \tag{101}$$

which is equal to 50*. The maximum is attained for* $x = 50$*,* $y = 0$ *and* $y' = 50$*.*

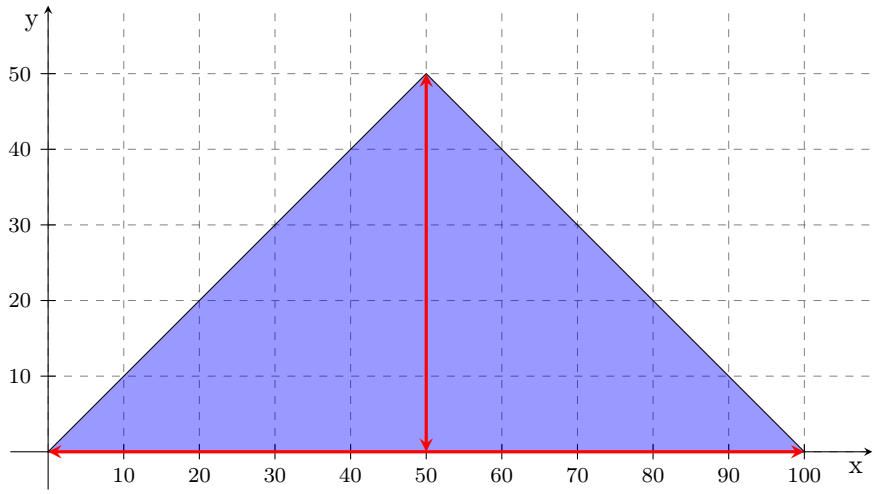

Figure 30: Two-dimensional set with sizes in x and y directions indicated by red double-headed arrows

7.5.3 Avoiding loop coalescing in the Pluto scheduler

To prevent near coalescing (93) from occurring, the constraint

$$|c_i| \leq \left\lceil \frac{S_j}{2} \right\rceil |c_j| \tag{102}$$

could be imposed on all pairs of schedule coefficients. However, since c_i may attain both positive and negative values, this is actually a disjunctive constraint, depending on the sign of c_i. The `isl` scheduler therefore imposes the stricter constraint

$$|c_i| \leq \left\lceil \frac{S_j}{2} \right\rceil \tag{103}$$

for each pair of schedule coefficients, or simply

$$|c_i| \leq \min_{j \neq i} \left\lceil \frac{S_j}{2} \right\rceil \tag{104}$$

for each schedule coefficient c_j. This is similar to imposing a fixed bound on the schedule coefficients, except that it is only imposed when the instance set is bounded by a fixed constant and that the bound imposed on the schedule coefficients is derived from the sizes of the instance set. This means that skewing by a factor of, say, 2, is allowed as long as the sizes in all directions are sufficiently large, but disallowed when one of those sizes is very small. The disadvantage of using bound (104) instead of bound (102) is that skewing by a small factor is disallowed as soon as the instance set has a small size in *any* coordinate direction.

7.5.4 Recovering from loop coalescing in the Feautrier scheduler

As mentioned in Section 6.6.2, no bound can be imposed on the schedule coefficients as in constraint (104). The reason is that the Feautrier scheduler (initially) solves an LP problem rather than an ILP problem. If the computed schedule coefficients are not integers, then the numerators with respect to a common denominator are used as the actual schedule coefficients. Imposing a

bound on the rational schedule coefficients does not help to prevent large values in the numerators themselves or in ratios among those numerators. Such a bound would therefore not prevent coalescing. Furthermore, imposing such a bound would invalidate Feautrier (1992b, Lemma 5), meaning that the variables e_i introduced in Section 6.4.1 could attain values different from 0 or 1.

Instead of trying to *prevent* coalescing in the Feautrier scheduler, the `isl` scheduler therefore tries to *detect* coalescing in the result and to *recover* from such coalescing. In particular, after computing the schedule coefficients, the `isl` scheduler checks whether condition (93), i.e.,

$$|c_i| > \left\lceil \frac{S_j}{2} \right\rceil |c_j| \tag{105}$$

holds for any pair of schedule coefficients. If so, schedule coefficient c_j is forced to zero and the LP problem is solved again. Since c_j is zero is the new solution, the same coalescing no longer occurs, but coalescing may still occur for other pairs of schedule coefficients. This process continues until there is no more coalescing or until no more solutions can be found. In the latter case, the previous solution is used as the result. This previous solution will still have coalescing, but since the Feautrier scheduler is used as a fallback, it is better to have a schedule with undesirable features than to have no schedule at all. The switch to an ILP problem described in Section 6.4.3 is only performed after all coalescing has been removed and only if the solution with coalescing removed is non-integral. Note that prior to version `isl-0.18-712-g484bccb`, the coalescing detection was based on (92) rather than on (93).

Example 30 *Continuing from Example 13 on page 39, the affine schedule function* (68)*, i.e.,*

$$\{ \mathtt{S}[i,j] \to 10000\, i + j \}, \tag{106}$$

satisfies condition (105) *with* $c_i = 10000$*,* $c_j = 1$ *and* $S_j = 10000$*. Schedule coefficient* c_j *is therefore forced to be equal to zero and a new solution is computed, resulting in*

$$\{ \mathtt{S}[i,j] \to i \}. \tag{107}$$

Condition (105) *is no longer satisfied and the solution is accepted.*

An alternative to recovering from coalescing would be to try and adjust the schedule constraints in order to prevent coalescing. This could be done by looking at all circuits in the dependence graph of validity schedule constraints and relaxing those constraints in the self-dependence schedule constraints along those circuits that could be abused to perform coalescing. However, experiments with a preliminary implementation of this idea showed it to be much slower than the current scheme of adjusting the schedule afterwards. It is however possible to drop at least *some* constraints to reduce the risk of coalescing, as described next.

7.5.5 Dropping coalescing constraints

In some cases where the Feautrier scheduler fallback gets used, it is glaringly obvious that the only sensible carrying schedule corresponds to a loop in the original program because there is a schedule constraint between an instance of that loop and the next instance of the same loop and this for every instance of every nested loop. Yet, if the instance sets are bounded by a fixed constant, then the Feautrier scheduler may still be able to find a coalescing affine schedule, which then needs to be rectified in a second call to the Feautrier scheduler with some coefficient set to zero. This section describes a particular instance where such

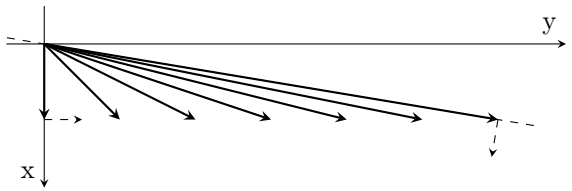

Figure 31: A graphical representation (in full arrows) of the schedule constraint difference set (110); the dashed arrows represent extreme valid directions

constraints can be removed beforehand, avoiding the need to call the Feautrier scheduler twice. This removal was introduced in `isl-0.18-724-gf936fb0ca5`.

Recall from Section 6.4.2 that for schedule constraints that relate a statement to itself, the constraints on the schedule coefficients are derived from the difference set of the schedule constraints. Any constraint in this difference set of the form

$$\delta_i \leq S_i \tag{108}$$

or

$$\delta_i \geq -S_i, \tag{109}$$

with S_i the size of the instance set in direction i as described in Section 7.5.2, can only be exploited to perform loop coalescing. They can therefore be removed without losing any interesting schedule solutions. The `isl` scheduler removes these constraints in both its Pluto scheduler implementation and in the Feautrier scheduler fallback.

Example 31 *Consider a difference set on schedule constraints of the form*

$$\{\, [1, b] : 0 \leq b \leq 6 \,\} \tag{110}$$

as illustrated in Figure 31 on page 63 and assume that the size in the second coordinate is equal to 6*. This means that there is a schedule constraint between pairs of instances where the first coordinate of the second instance is one higher than that of the first instance, while the second coordinate of the second instance is simply greater than or equal to the second coordinate of the first instance. As also illustrated in the figure, any linear constraint that is valid for all elements in the difference set and that has a negative coefficient for the second coordinate, needs to have a large coefficient for the first coordinate. In particular, applying the Farkas lemma to the difference set* (110) *yields*

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, d] : d + c_0^{\mathrm{x}} \geq 0 \wedge d + c_0^{\mathrm{x}} + 6\, c_1^{\mathrm{x}} \geq 0 \,\}. \tag{111}$$

The coefficient of the constant term d is only used to determine whether the schedule constraints are carried (with value 0 *or* -1*), so if c_1^{x} has a negative value, then c_0^{x} must have a positive value that is at least* 6 *times as large. Dropping the size constraint $b \leq 6$ from the difference set* (110) *results in*

$$\{\, [1, b] : 0 \leq b \,\} \tag{112}$$

and applying the Farkas lemma to this set yields

$$\{\, [c_0^{\mathrm{x}}, c_1^{\mathrm{x}}, d] : d + c_0^{\mathrm{x}} \geq 0 \wedge c_1^{\mathrm{x}} \geq 0 \,\}. \tag{113}$$

That is, the coefficient c_1^{x} is now forced to be non-negative, but that does not remove any interesting schedules since any schedule with a negative value for this coefficient is necessarily coalescing.

If both lower bound (109) and upper bound (108) appear in the difference set of the schedule constraints for a particular statement, then they are both removed. If those are the only constraints that involve the distance variable in that direction, then, after removal, the distance variable is completely unconstrained. This means that the corresponding schedule coefficient is forced to be zero. This coefficient can therefore be set to zero in all the constraints that are added to the LP problem constructed by the Feautrier scheduler. The same effect can be obtained by modifying the *input* to the application of the Farkas lemma, by eliminating the corresponding variable use Fourier-Motzkin elimination.

More generally, take the *lineality space* (Schrijver 1986, Chapter 8) of the difference set of the schedule constraints, i.e., the directions in which this difference set extends to both positive and negative infinity. These directions are called *lines*. The lineality space is defined by linear equality constraints

$$E\mathbf{x} = \mathbf{0}. \tag{114}$$

Elements $\mathbf{x}$ that have the same value for E lie on the same line and should be considered equivalent. Define the corresponding equivalence relation

$$R = \{ \mathbf{x} \to \mathbf{x}' : E\mathbf{x} = E\mathbf{x}' \}. \tag{115}$$

Composing the set of schedule constraints with this equivalence relation (on both sides) removes the distinction between elements on the same line. That is, if there is a schedule constraint between $\mathbf{x}$ and $\mathbf{x}'$, then after composition, there will be a schedule constraint between any point on a line containing $\mathbf{x}$ and any point on a line containing $\mathbf{x}'$. For intra-node schedule constraints, the equivalence relation R (115) can be applied to the difference set (once). Note that the lineality space is strictly speaking defined on a convex set. If the difference set is not convex, then the lineality space of its closed convex hull should be computed. In practice, `isl` computes the affine hull of the lineality spaces of the disjuncts in the disjunctive normal form of the set. Also note that this technique does not assume that the lineality space is oriented along the coordinate directions. A lineality space that is not oriented in this way will not result in a zero schedule coefficient, but rather in an equality constraint among two or more schedule coefficients. However, the lineality spaces created by the removal of lower bounds (109) and upper bounds (108) are always oriented along coordinate directions. The lineality space is taken into account since `isl-0.18-730-gd66283697d`.

Example 32 *Assume that after removal of coalescing bound constraints, the difference set on schedule constraints is of the form*

$$\{ [a, b, c] : a > 0 \vee (a = 0 \wedge b > 0) \vee (a = 0 \wedge b = 0 \wedge c > 0) \}. \tag{116}$$

The lineality spaces for each disjunct separately are

$$\{ [a, b, c] : a = 0 \}, \tag{117}$$

$$\{ [a, b, c] : a = 0 \wedge b = 0 \} \tag{118}$$

and

$$\{ [a, b, c] : a = 0 \wedge b = 0 \wedge c = 0 \}. \tag{119}$$

The combined lineality space is

$$\{ [a, b, c] : a = 0 \}. \tag{120}$$

```
for (int i = 0; i < 10; ++i) {
A:      A[i] = B[i];
B:      B[i + 1] = A[0];
}
```

Listing 32: Constraint removal example

The equivalence relation (115) *is therefore*

$$\{\,[x_0, x_1, x_2] \to [x_0', x_1', x_2'] : x_0 = x_0'\,\}. \tag{121}$$

Applying this relation to the difference set (116) *yields*

$$\{\,[a, b, c] : a > 0 \vee a = 0\,\}. \tag{122}$$

Note that the second and the third disjunct are the same after this application and is only shown once. This common disjunct could in theory be combined with the first disjunct to form $a \geq 0$*, but this may result in the Feautrier scheduler being unable to carry any schedule constraints since it is able to carry the constraints of the form* $a > 0$*, but not those of the form* $a = 0$ *or* $a \geq 0$*.*

Since removing constraints from the difference set of schedule constraints can help to prevent the construction of coalescing schedules, it may be tempting to remove all constraints that are derived from the instance sets and to only preserve the constraints that express the connection between the two instances involved. That is, the domain and the range of the relation containing the schedule constraints could be simplified with respect to the instance set. Removing these constraints does indeed reduce the risk of coalescing, but this removal is too indiscriminate and does not only remove undesirable combinations of schedule constraints, but may also remove good solutions to the point of even removing all solutions in extreme cases.

Example 33 *Consider the slightly contrived code fragment shown in Listing 32 on page 65. The instance set is*

$$\{\,\mathtt{A}(i) : 0 \leq i \leq 9; \mathtt{B}(i) : 0 \leq i \leq 9\,\}, \tag{123}$$

while the flow dependence relation is

$$\{\,\mathtt{A}(0) \to \mathtt{B}(i) : 0 \leq i \leq 9; \mathtt{B}(i) \to \mathtt{A}(1 + i) : 0 \leq i \leq 8\,\}. \tag{124}$$

This flow dependence relation is shown in Figure 33 on page 66. Using these flow dependences as validity schedule constraints results in a schedule that is equivalent to the original execution order. Simplifying the flow dependence relation with respect to the instance set results in

$$\{\,\mathtt{A}(0) \to \mathtt{B}(i); \mathtt{B}(i) \to \mathtt{A}(1 + i)\,\}. \tag{125}$$

Note that $\mathtt{A}(0)$ *is now not just related to any* $\mathtt{B}(i)$ *with* $0 \leq i \leq 9$*, but simply to* any $\mathtt{B}(i)$*, including those with negative values of* i*. Using these simplified flow dependences as validity schedule constraints no longer admits any schedule. This can easily be seen by looking at the composition of relation* (125) *with itself, i.e.,*

$$\{\,\mathtt{A}(0) \to \mathtt{A}(i); \mathtt{B}(-1) \to \mathtt{B}(i)\,\}. \tag{126}$$

That is, $\mathtt{A}(0)$ *needs to be scheduled before any* $\mathtt{A}(i)$*, including* $\mathtt{A}(0)$ *itself.*

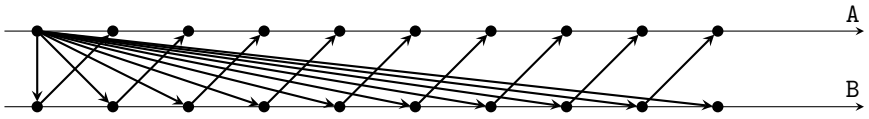

Figure 33: The flow dependences for the code fragment in Listing 32

7.6 Self-dependences first

Recall from Section 6.6.4 that the Feautrier scheduler fallback has a tendency to carry more schedule constraints than strictly needed for its use by the `isl` scheduler, The unscaling of Section 7.1 tries to *recover* from the scaling that is often introduced as a result of this tendency. This section presents a simple but effective technique for *preventing* needless carrying of schedule constraints. This means that in those cases where this latter technique is effective, the unscaling of Section 7.1 is no longer needed. The technique was introduced in `isl-0.18-713-gf0e0f1f47a`.

As explained in Section 7.1, the spurious scaling introduced by the Feautrier scheduler is caused by its desire to carry schedule constraints between pairs of successive statements. If the `--schedule-carry-self-first` option is set (enabled by default), the `isl` scheduler therefore first tries to set up an LP problem that only carries groups of schedule constraints from a node to itself. That is, variables e_i that are used to reflect whether a group of schedule constraints is carried are only introduced for such groups of schedule constraints from a node to itself. Clearly, the other groups need to be taken into account as well, but these constraints only need to be satisfied. That is, for groups of schedule constraints between different nodes, the schedule coefficient constraints (20) are replaced by

$$g(-(\mathbf{c}_j^{\mathrm{x},+} - \mathbf{c}_j^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{x},+} - \mathbf{c}_k^{\mathrm{x},-}), (\mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}}), c_k^{\mathrm{c}} - c_j^{\mathrm{c}}; \mathbf{x}, \mathbf{y}) \geq 0. \tag{127}$$

Note that only trying to carry groups of schedule constraints between a node and itself may in some cases not result in a non-trivial schedule. This happens in particular when there are no such groups. If a trivial solution is obtained, the scheduler therefore tries again taking into account all groups of schedule constraints for carrying.

Example 34 *Recall from Example 16 on page 41 that when the Feautrier scheduler attempts to carry all groups (i.e., when the `--schedule-carry-self-first` option is not set), the affine schedule it computes as outer band for the code fragment in Listing 14 on page 27 is*

$$\mathtt{A}[i,j,k] \to i + j + k; \mathtt{B}[i,j] \to i + 2j; \mathtt{C}[i,k] \to 2i + k; \mathtt{D}[i] \to 3i. \tag{128}$$

When the option is set, it computes the affine schedule

$$\mathtt{A}[i,j,k] \to k; \mathtt{B}[i,j] \to j; \mathtt{C}[i,k] \to k; \mathtt{D}[i] \to i \tag{129}$$

instead. This is the same as the schedule computed by the alternative approach of Section 6.7.2 as illustrated in Example 18 on page 43.

First trying to carry only groups of schedule constraints between a node and itself does not only prevent some spurious scalings, it can also prevent some spurious shifts, as illustrated in the next example.

```
A:y[0] = -r[0];
B:beta = SCALAR_VAL(1.0);
C:alpha = -r[0];

 for (k = 1; k < _PB_N; k++) {
D: beta = (1-alpha*alpha)*beta;
E: sum = SCALAR_VAL(0.0);
   for (i=0; i<k; i++) {
F:    sum += r[k-i-1]*y[i];
   }
G: alpha = - (r[k] + sum)/beta;

   for (i=0; i<k; i++) {
H:    z[i] = y[i] + alpha*y[k-i-1];
   }
   for (i=0; i<k; i++) {
I:   y[i] = z[i];
   }
J: y[k] = alpha;
 }
```

Listing 34: Excerpt from `PolyBench/C 4.1` `durbin` benchmark, with additional statement labels

`isl` version	number of groups		time
`isl-0.18-674-g954b422b25`		2335	1m13.496s
`isl-0.18-675-g757d4cec0c`		248	0m28.404s
`isl-0.18-712-g484bccbf98`		248	0m27.584s
	intra	inter	
`isl-0.18-713-gf0e0f1f47a`	86	162	0m18.632s
`isl-0.18-724-gf936fb0ca5`	86	162	0m15.592s
`isl-0.18-730-gd66283697d`	74	152	0m13.456s

Table 37: Effect of various changes on the number of groups of schedule constraints and the `isl_schedule` execution time for the code from Example 25 on page 55

Example 35 *Consider the code fragment show in Listing 34 on the following page. Without setting the `--schedule-carry-self-first` option, the schedule shown in Listing 35 is produced. Note that the `I`-statement is shifted by one in the outer band. The only reason for this shift is that it happens to result in one more carried group of schedule constraints between the `I`-statement and some other statement. With `--schedule-carry-self-first`, the scheduler can focus on the essential parts and produce the schedule shown in Listing 36 on page 69.*

Example 36 *Table 37 on page 67 shows the effect of various changes that may affect the number of groups of schedule constraints considered by the Feautrier scheduler or the representation of these constraints, including the improvement described in the current section. The input is formed by the schedule constraints obtained from the code from Example 25 on page 55, after the grouping of Section 7.4. The reported time is the total execution time of `isl_schedule` with option `--schedule-outer-coincidence`, which is turned on by default by `PPCG`,*

```
domain: "[n] -> { A[]; E[k] : 0 < k < n; C[]; G[k] : 0 < k < n; B[];
        F[k, i] : k < n and 0 <= i < k;
        H[k, i] : k < n and 0 <= i < k; J[k] : 0 < k < n;
        I[k, i] : k < n and 0 <= i < k; D[k] : 0 < k < n }"
child:
  sequence:
  - filter: "[n] -> { C[] }"
  - filter: "[n] -> { B[] }"
  - filter: "[n] -> { A[] }"
  - filter: "[n] -> { E[k]; G[k]; H[k, i]; F[k, i]; J[k]; I[k, i]; D[k] }"
    child:
      schedule: "[n] -> [{ E[k] -> [(k)]; G[k] -> [(k)];
        H[k, i] -> [(k)]; F[k, i] -> [(k)]; J[k] -> [(k)];
        I[k, i] -> [(1 + k)]; D[k] -> [(k)] }]"
      child:
        sequence:
        - filter: "[n] -> { I[k, i] }"
          child:
            schedule: "[n] -> [{ I[k, i] -> [(i)] }]"
            permutable: 1
            coincident: [ 1 ]
        - filter: "[n] -> { E[k] }"
        - filter: "[n] -> { F[k, i] }"
          child:
            schedule: "[n] -> [{ F[k, i] -> [(i)] }]"
        - filter: "[n] -> { D[k] }"
        - filter: "[n] -> { G[k] }"
        - filter: "[n] -> { H[k, i] }"
          child:
            schedule: "[n] -> [{ H[k, i] -> [(i)] }]"
            permutable: 1
            coincident: [ 1 ]
        - filter: "[n] -> { J[k] }"
```

Listing 35: Schedule for the code in Listing 34 when the Feautrier scheduler is instructed to carry all groups of schedule constraints

```
domain: "[n] -> { A[]; E[k] : 0 < k < n; C[]; G[k] : 0 < k < n; B[];
        F[k, i] : k < n and 0 <= i < k;
        H[k, i] : k < n and 0 <= i < k; J[k] : 0 < k < n;
        I[k, i] : k < n and 0 <= i < k; D[k] : 0 < k < n }"
child:
  sequence:
  - filter: "[n] -> { C[] }"
  - filter: "[n] -> { B[] }"
  - filter: "[n] -> { A[] }"
  - filter: "[n] -> { E[k]; G[k]; H[k, i]; F[k, i]; J[k]; I[k, i]; D[k] }"
    child:
      schedule: "[n] -> [{ E[k] -> [(k)]; G[k] -> [(k)];
        H[k, i] -> [(k)]; F[k, i] -> [(k)]; J[k] -> [(k)];
        I[k, i] -> [(k)]; D[k] -> [(k)] }]"
      child:
        sequence:
        - filter: "[n] -> { E[k] }"
        - filter: "[n] -> { F[k, i] }"
          child:
            schedule: "[n] -> [{ F[k, i] -> [(i)] }]"
        - filter: "[n] -> { D[k] }"
        - filter: "[n] -> { G[k] }"
        - filter: "[n] -> { H[k, i] }"
          child:
            schedule: "[n] -> [{ H[k, i] -> [(i)] }]"
            permutable: 1
            coincident: [ 1 ]
        - filter: "[n] -> { I[k, i] }"
          child:
            schedule: "[n] -> [{ I[k, i] -> [(i)] }]"
            permutable: 1
            coincident: [ 1 ]
        - filter: "[n] -> { J[k] }"
```

Listing 36: Schedule for the code in Listing 34 on page 67 when the Feautrier scheduler is instructed to only carry self-groups

but not by `isl`*. This total execution time includes the execution time of detecting that no coincident member can be found at the outer level, the time of the Feautrier scheduler fallback (where the number of groups is measured) and the time to complete the schedule.*

The first row of the table corresponds to the version of `isl` *right before the removal of duplicate disjuncts described in Section 6.4.4. The next row corresponds to the version that introduced this removal. The third row corresponds to the version of* `isl` *right before the introduction of the mechanism described in this section, while the fourth row corresponds to this introduction. Starting from the fourth row, the number of groups is split into those between a node and itself and those between distinct nodes. Only the first set ends up getting used in this example because the self-dependences are sufficient for computing a non-trivial schedule. The fifth row shows the effect of the dropping of constraints from Section 7.5.5, while the sixth row shows the effect of exploiting the lineality space described in the same section.*

References

Acharya, Aravind and Uday Bondhugula (2015). "PLUTO+: Near-complete Modeling of Affine Transformations for Parallelism and Locality". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA: ACM, pp. 54–64. DOI: 10.1145/2688500.2688512. [35, 37, 38]

Akilesh, B (Mar. 2017). *Message on* `pluto-development@googlegroups.com`. [58]

Alias, Christophe, Alain Darte, and Alexandru Plesco (June 2011). *Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis*. Tech. rep. RR-7648. INRIA. [18]

Baghdadi, Riyadh, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Róbert Dávid, and Elnar Hajiyev (Oct. 2015). "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming". In: *Proc. Parallel Architectures and Compilation Techniques (PACT'15)*. DOI: 10.1109/PACT.2015.17. [3, 9]

Balev, Stephan, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset (1998). "Linear programming models for scheduling systems of affine recurrence equations – a comparative study". In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, pp. 250–258. DOI: http://doi.acm.org/10.1145/277651.277691. [23]

Bondhugula, Uday (2008). "Effective automatic parallelization and locality optimization using the polyhedral model". PhD thesis. Columbus, OH, USA. [46]

Bondhugula, Uday, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan (Apr. 2008). "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC)*. DOI: 10.1007/978-3-540-78791-4_9. [20, 34, 35, 37, 45]

Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan (2008). "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming lan-*

guage design and implementation. PLDI '08. Tucson, AZ, USA: ACM, pp. 101–113. DOI: 10.1145/1375581.1375595. [39]

Detlefs, David, Greg Nelson, and James B. Saxe (2005). "Simplify: a theorem prover for program checking". In: *J. ACM* 52.3, pp. 365–473. DOI: 10.1145/1066100.1066102. [35]

Feautrier, Paul (1988a). "Array expansion". In: *ICS '88: Proceedings of the 2nd international conference on Supercomputing*. St. Malo, France: ACM Press, pp. 429–441. DOI: 10.1145/55364.55406. [13]

Feautrier, Paul (1988b). "Parametric Integer Programming". In: *RAIRO Recherche Opérationnelle* 22.3, pp. 243–268. [24, 25, 35]

Feautrier, Paul (1991). "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1, pp. 23–53. DOI: 10.1007/BF01407931. [5]

Feautrier, Paul (Oct. 1992a). "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time". In: *International Journal of Parallel Programming* 21.5, pp. 313–348. DOI: 10.1007/BF01407835. [22, 23]

Feautrier, Paul (Dec. 1992b). "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6, pp. 389–420. DOI: 10.1007/BF01379404. [2, 20, 23, 24, 62]

Feautrier, Paul, Jean-François Collard, and Cédric Bastoul (2003). *Solving Systems of Affine (In)Equalities: PIP's User's Guide*. Tech. rep. PRiSM, Versailles University.

Feautrier, Paul and Christian Lengauer (2011). "The Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, pp. 1581–1592. [2]

Grosser, Tobias, Armin Größlinger, and Christian Lengauer (2012). "Polly - Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04. DOI: 10.1142/S0129626412500107. [39, 51]

Grosser, Tobias, Sven Verdoolaege, and Albert Cohen (July 2015). "Polyhedral AST generation is more than scanning polyhedra". In: *ACM Transactions on Programming Languages and Systems* 37.4, 12:1–12:50. DOI: 10.1145/2743016. [4]

Ho, Minh Quan (Feb. 2017). *Personal communication*. [55]

Juega, Juan Carlos (Dec. 2011). *Personal communication*. [47]

Lim, Amy W. and Monica S. Lam (1997). "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms". In: *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*. Paris, France: ACM Press, pp. 201–214. DOI: 10.1145/263699.263719. [21, 23]

Lim, Amy W., Shih-Wei Liao, and Monica S. Lam (2001). "Blocking and array contraction across arbitrarily nested loops using affine partitioning". In: *ACM SIGPLAN Notices* 36.7, pp. 103–112. DOI: 10.1145/568014.379586. [47]

Makhorin, Andrew (n.d.). *GLPK (GNU Linear Programming Kit)*. [38]

Mehta, Sanyam, Pei-Hung Lin, and Pen-Chung Yew (2014). "Revisiting Loop Fusion in the Polyhedral Framework". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, pp. 233–246. DOI: 10.1145/2555243.2555250. [46]

Mehta, Sanyam and Pen-Chung Yew (2015). "Improving Compiler Scalability: Optimizing Large Programs at Small Price". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementa-*

tion. PLDI 2015. Portland, OR, USA: ACM, pp. 143–152. DOI: 10.1145/2737924.2737954. [51]

Meister, Benoît (Dec. 2004). "Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization". PhD thesis. Université Louis Pasteur. [44]

Nvidia (2011). *Nvidia CUDA C Programming Guide 4.0*. [2]

Obrecht, Christian, Bernard Tourancheau, and Frédéric Kuznik (Sept. 2015). "Performance Evaluation of an OpenCL Implementation of the Lattice Boltzmann Method on the Intel Xeon Phi". In: *Parallel Processing Letters* 25.3. DOI: 10.1142/S0129626415410017. [55]

Polychronopoulos, Constantine D. (1987). "Loop coalescing: A compiler transformation for parallel machines". In: *International conference on parallel processing*. [39]

Pouchet, Louis-Noel (2012). *PolyBench/C 3.2*. [2, 46]

Pouchet, Louis-Noel and Tomofumi Yuki (2015). *PolyBench/C 4.1*. [2, 17, 21, 27, 33, 43, 49, 52–54, 67]

Pugh, William and David Wonnacott (1994). "An Exact Method for Analysis of Value-based Array Data Dependences". In: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, pp. 546–566. DOI: 10.1007/3-540-57659-2_31. [5]

Schrijver, Alexander (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons. [22, 32, 45, 64]

Stone, John E., David Gohara, and Guochun Shi (2010). "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering*. DOI: 10.1109/MCSE.2010.69. [2]

Trifunovic, Konrad, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta (2010). "GRAPHITE two years after: First lessons learned from real-world polyhedral compilation". In: *GCC Research Opportunities Workshop (GROW'10)*. [51]

Truong, Leonard, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman (2016). "Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, pp. 209–223. DOI: 10.1145/2908080.2908105. [58]

Vasilache, Nicolas, Benoît Meister, Muthu Baskaran, and Richard Lethin (Jan. 2012). "Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization". In: *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France. [35, 38]

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49. [2]

Verdoolaege, Sven (May 2015). *PENCIL support in pet and PPCG*. Tech. rep. RT-457, version 2. INRIA Paris-Rocquencourt. DOI: 10.13140/RG.2.1.4063.7926. [3, 4, 7, 9, 17]

Verdoolaege, Sven (2016). *Presburger Formulas and Polyhedral Compilation*. DOI: 10.13140/RG.2.1.1174.6323. [4, 23]

Verdoolaege, Sven and Albert Cohen (Jan. 2016). "Live-Range Reordering". In: *Proceedings of the sixth International Workshop on Polyhedral Compilation Techniques*. Prague, Czech Republic. DOI: 10.13140/RG.2.1.3272.9680. [2, 5, 7]

Verdoolaege, Sven and Tobias Grosser (Jan. 2012). "Polyhedral Extraction Tool". In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France. DOI: 10.13140/RG.2.1.4213.4562. [3]

Verdoolaege, Sven, Serge Guelton, Tobias Grosser, and Albert Cohen (Jan. 2014). "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria. DOI: 10.13140/RG.2.1.4475.6001. [13]

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713. [2, 35, 37, 47]

Zinenko, Oleksandr (Apr. 2017). *Personal communication*. [27]

Index